University of Zurich[UZH]

Department of Informatics

Software Runtime Analytics for Developers

Extending Developers' Mental Models by Runtime Dimensions

Dissertation submitted to the Faculty of Business,
Economics and Informatics
of the University of Zurich

to obtain the degree of
Doktor / Doktorin der Wissenschaften, Dr. sc.
(corresponds to Doctor of Science, PhD)

presented by
Jürgen Cito
from Vienna, Austria

approved in February 2018

at the request of
Prof. Dr. Harald C. Gall, University of Zurich, Switzerland
Dr. Philipp Leitner, Chalmers University of Technology, Sweden
Prof. Dr. Nenad Medvidovic, University of Southern California, USA

University of Zurich[UZH]

The Faculty of Business, Economics and Informatics of the University of Zurich hereby authorizes the printing of this dissertation, without indicating an opinion of the views expressed in the work.

Zurich, February 14th, 2018

Chairwoman of the Doctoral Board: Prof. Dr. Sven Seuken

Abstract

Software systems have become instrumental in almost every aspect of modern society. The reliability and performance of these systems plays a crucial role in the every day lives of their users. The performance of a software system is governed by its program code, executed in an environment. To ensure reliability and performance, developers need to understand both code and its execution effects in the context of this environment. Reasoning about source code of programs is a complex cognitive process in which developers construct mental models that are informed by their knowledge of control- and data-flow. Reasoning about runtime aspects of code (e.g., performance) requires consulting external information sources, such as profilers, to construct a more complete image. For software deployed in scalable cloud infrastructures, developers gain insight into operational aspects of code by inspecting distributed runtime data (logs, traces, metrics) to reason about the peculiarities of production environments. Runtime data is currently presented in dashboards that display metrics in time series graphs and enable search capabilities for properties of the data. In a mixed-method empirical study with software developers who deploy in cloud infrastructures, we found that this particular presentation of data is an obstacle to gain actionable insights for software maintenance and debugging tasks. Developers end up relying on their beliefs and intuition about the system to introduce potential fixes, rather than making informed decisions based on data. We propose an approach called *Software Runtime Analytics for Developers* that integrates operational aspects from production into the daily workflow of software developers, thereby extending their existing mental models of programs by runtime dimensions, and enables

data-driven decision making for developers. Specifically, we design a framework that is an abstract representation of potential solution spaces that can guide implementations of this approach.

We instantiate two concrete solutions out of the abstract framework and implement prototypes that serve as proof-of-concepts. *PerformanceHat* is an Eclipse IDE plugin that models runtime performance data and matches it to specific, performance-related elements of source code in the IDE (methods, loops, and collections). Given the information on the source code level, we leverage fast inference models that provide live feedback of performance properties of newly written and modified code to prevent performance problems from reaching production. *Context-based Analytics* is a web application in Python that supports problem diagnosis by establishing explicit links between runtime data and program code fragments. The resulting graph allows developers to navigate the solution space and relate problems at runtime to their origin in source code.

We evaluated *PerformanceHat* in a controlled experiment with 20 professional software developers, in which they worked on software maintenance tasks using our approach and a representative baseline (Kibana). We found that using our approach, developers are significantly faster in (1) detecting the performance issue, and (2) finding the root-cause of the issue. We also let both groups work on non-performance relevant tasks and found no significant differences in task time. We conclude that our approach helps detect, prevent, and debug performance problems faster, while at the same time not adding a disproportionate distraction for maintenance tasks not related to performance. For *Context-based Analytics* we conducted a case study within IBM to evaluate to what extent our approach can be used to diagnose runtime issues in real-life applications. We designed a study that compares the problem diagnosis process for two issues taken from the issue tracker of an application in IBM Bluemix and found that our approach reduced the effort of diagnosing these issues. In particular, it decreased the number of required analysis steps by 48% and the number of needed inspected traces by 40% on average as compared to a standard diagnosis approach within the application team (Kibana).

Zusammenfassung

Softwaresysteme haben eine tragende Rolle in fast jedem Aspekt der modernen Gesellschaft. Die Zuverlässigkeit und Performance dieser Systeme spielt eine entscheidende Rolle im täglichen Leben ihrer Benutzer. Die Performance eines Softwaresystems ist abhängig von seinem Programmcode, der in einer Umgebung ausgeführt wird (beispielsweise in der Cloud). Um Zuverlässigkeit und Performance zu gewährleisten, müssen Entwickler sowohl Code als auch dessen Auswirkungen in der Umgebung verstehen. Das Verstehen des Quellcodes von Programmen ist ein komplexer kognitiver Prozess, bei dem Entwickler mentale Modelle konstruieren, die durch ihr Wissen über Kontroll- und Datenflüsse gesteuert wird. Um Laufzeitaspekte von Code (z.B. Performance) zu verstehen müssen externe Informationsquellen, beispielsweise Profiler, in Betracht gezogen werden. Bei Software, die in skalierbaren Cloud-Infrastrukturen deployed wird, können Entwickler nur dann Laufzeitaspekte des Codes in Erfahrung bringen indem sie verteilte Laufzeitdaten (Logs, Traces, Metriken) untersuchen, um die Besonderheiten von komplexen Produktionsumgebungen zu verstehen. Laufzeitdaten werden in heutigen Systemen in Dashboards angezeigt, die Messwerte des Systems als Visualisierung von Zeitreihen anzeigen und Suchfunktionen für Eigenschaften der Daten ermöglichen. In einer empirischen Studie mit Software Entwicklern, die ihre Applikationen in Cloud-Infrastrukturen deployen, haben wir festgestellt, dass diese spezielle Darstellung von Daten ein Hindernis für verwertbare Erkenntnisse für Softwarewartungs- und Debugging-Aufgaben darstellt. Entwickler verlassen sich letztendlich auf ihre Intuition über das System um mögliche Korrekturen einzuführen, anstatt fundierte Entscheidungen

auf Basis von Daten zu treffen. Wir schlagen einen Ansatz namens *Software Runtime Analytics for Developers* vor, der Laufzeit Aspekte aus der Produktion in den täglichen Workflow von Softwareentwicklern integriert, und damit die vorhandenen mentalen Modelle um Laufzeitdimensionen erweitert und damit datengestützte Entscheidungsfindung für Entwickler ermöglicht. Insbesondere entwerfen wir ein Framework, der eine abstrakte Darstellung möglicher Lösungsräume repräsentiert, der die Implementierungen dieses Ansatzes leiten können.

Wir instanziieren zwei konkrete Lösungen aus dem abstrakten Framework und implementieren Prototypen, die als Proof-of-Concepts dienen. *PerformanceHat* ist ein Eclipse-IDE-Plug-in, das Laufzeitdaten modelliert und mit bestimmten Elementen des Quellcodes in der IDE (Methoden, Schleifen, und Collections) abgleicht. Auf Basis der Informationen auf Ebene des Quellcodes verwenden wir schnelle Inferenzmodelle, die eine Live-Rückmeldung der Performance Eigenschaften von neu geschriebenem und geändertem Code bieten, um zu verhindern, dass Performance Probleme in Produktion deployed werden. *Context-based Analytics* ist eine Webanwendung in Python, die die Problemdiagnose unterstützt, indem explizite Verknüpfungen zwischen Laufzeitdaten und Programmcode Fragmenten hergestellt werden. Der resultierende Graph ermöglicht es Entwicklern, im Lösungsraum zu navigieren und Probleme zur Laufzeit mit ihrem Ursprung im Quellcode in Beziehung zu setzen. Wir haben *PerformanceHat* in einem kontrollierten Experiment mit 20 professionellen Entwicklern evaluiert, in dem sie mit unserem Ansatz und einer repräsentativen Baseline (Kibana) an Softwarewartungsaufgaben gearbeitet haben. Wir haben festgestellt, dass Entwickler mit unserem Ansatz wesentlich schneller (1) das Performance Problem erkennen und (2) die Ursache des Problems finden. Wir ließen beide Gruppen an nicht-performance relevanten Aufgaben arbeiten und fanden keine signifikanten Unterschiede in der Zeit in denen die Aufgaben erledigt wurden. Wir kommen zu dem Schluss, dass unser Ansatz hilft Performance Probleme schneller zu erkennen, zu verhindern und zu beheben, während gleichzeitig Wartungsaufgaben, die nicht mit Performance verbunden sind, keine unverhältnismäßige Ablenkung in den Prozess einführen. Für *Context-based Analytics* haben wir in einer Fallstudie

innerhalb von IBM untersucht, inwieweit unser Ansatz zur Diagnose von Laufzeitproblemen in realen Anwendungen geeignet ist. Wir haben eine Case Study entwickelt die den Problemdiagnoseprozess für zwei Probleme vergleicht, die aus dem Issue-Tracker einer Anwendung in IBM Bluemix stammen, und festgestellt, dass unser Ansatz den Aufwand für die Diagnose dieser Probleme verringert hat. Insbesondere wurde die Anzahl der erforderlichen Analyseschritte um 48% und die Anzahl der benötigten Inspektion von Laufzeitdaten um durchschnittlich 40%, im Vergleich zum Diagnosetool Kibana verringert.

Contents

List of Figures

List of Tables

1

Synopsis

Software systems have become instrumental in almost every aspect of life and modern society at large. They are the backbone of our transportation and logistics systems, communication systems, financial markets, commerce, modern medicine and many more. This, albeit incomplete but illustrative, list demonstrates the pervasive nature of software. The reliability and performance of these systems plays a crucial role in the every day lives of their users. The effects of poor software performance can range from mild frustration when shopping online to much more drastic ramifications in the case of medical imaging. The performance of a software system is always a function of its program code and execution environment. Developing and maintaining these systems is a complex, human activity. Developers write program code on their local workstations that eventually gets deployed to production environments, so that it can be accessed by users or other programs. To ensure reliability and performance, we need to make sure developers properly understand the complexities of their program code and execution environments.

Mental Models of Understanding Programs. To effectively create and maintain programs, developers have to understand its components and their interrelationships. The process of understanding program code is called *program comprehension*. Program comprehension is a complex cognitive process involving the construction of a mental representation of structure and execution of software [Von Mayrhauser and Vans, 1995]. Acquiring a mental model of a program involves developing a knowledge structure representation based on control flow and the relationship between these structures based on data flow [Pennington, 1987a, Pennington, 1987b]. Understanding and improving runtime properties, such as performance, additionally requires developers to have information on the operational aspects of their software during execution. Hence, a crucial factor to ensure the performance and reliability of software relies heavily on having insight into the behavior of software at runtime.

Insight through Observation. To gain a more comprehensive picture of the operational aspects of programs, we need to extract information of the inner workings of programs at runtime, which is one of the pillars of *software performance engineering (SPE)*. SPE is concerned with a broad set of activities that support developers to meet performance requirements of their programs throughout the software development life-cycle [Smith, 1993]. In particular, it deals with methods of measurement and models to understand and improve performance properties of software systems [Woodside et al., 2007]. Software operations engineers are usually tasked to facilitate the measurement aspects of SPE by capturing the interactions and behavior of software. This is achieved by observing runtime systems in production through monitoring and instrumentation. More concretely, application and infrastructure logs and metrics on server instances are collected and stored in a centralized storage from which the aggregated data is populated in dashboards. They visualize data distributions in the form of time series graphs and enable search capabilities for properties of runtime data.

However, acquiring and visualizing this data does not yet guarantee that developers integrate it into their development and debugging workflows and

processes. We briefly characterize the high-level steps of identifying runtime issues to illustrate its challenges.

The Dilemma of Moving Fast. Many performance and reliability problems that surface in production often originate in source code [Hamilton, 2007]. Developers then have the challenging task to reason about runtime behavior and determine why certain parts of their code do not exhibit desired properties in production. Due to the complex nature of cloud deployments, which are distributed systems with scalable and ephemeral infrastructure [Mell et al., 2011, Cito et al., 2015b], local profilers do not reveal the right information to solve certain classes of performance issues. To complement their mental models with the operational aspects of their code, developers often have to inspect different sources of runtime information from production environments. We conjecture that extending developer's mental models with quantitative runtime information during software development and debugging workflows provides actionable insight that prevents these runtime issues from being introduced in the first place:

> ***Hypothesis 1:*** *"Developers can prevent runtime issues from reaching production if their mental models are extended by operational aspects of their code from production runtime"*

The processes that are involved to ensure runtime properties meet certain non-functional requirements are time-consuming. It includes tasks such as capacity planning and large-scale performance testing. However, the need for rapidly delivering new functionality to survive in a highly-competitive market has been fueling the adoption of DevOps practices [Parnin et al., 2017]. Breaking the metaphorical barrier between software operations and development teams, introducing new methodologies, cultural changes, and automation tools allows teams to continuously release new code to production in cloud environments. In this "move fast" philosophy of software release management, there is rarely the time for rigorous performance testing or performance modeling of new features [Feitelson

et al., 2013]. However, when an application exhibits failures in production, this fast paced delivery cycle is suddenly halted. At odds with its initial intent, this extreme attitude in quickly deploying new features could potentially slow down the continuous delivery cycle, as problems might occur in higher frequencies, and take longer to resolve, the faster new code is deployed. Therefore, it is critical that for the issues mentioned above, developers are able to quickly associate the runtime problem with the relevant source code:

> ***Hypothesis 2:*** *"Developers are faster in diagnosing runtime issues when runtime information is explicitly linked to source code fragments from where the problem originated"*

Tracing these issues back to their origin in code often requires developers to inspect multiple fragments of information that have to be queried and analyzed in the aforementioned dashboards.

Challenges. The particular design and level of abstraction of runtime information in dashboards and similar tools is largely designed to enable the workflow of software operations, whose responsibilities require them to think in systems, rather than in source code. The system-centric perspective of operations analytics is not in line with the workflow of developers, who struggle to incorporate operational aspects of their code into their development and debugging activities. This goes along with the notion of existing work that argues that program analysis can only be effective if it is smoothly integrated into the development workflow [Sadowski et al., 2015]. We argue that the abstraction level at which runtime information is currently presented and integrated is not well-suited to support developers in understanding the operational complexities of their code and help them make data-driven decisions about potential code changes.

Statement and Goals. To summarize: Software performance engineering is concerned with the systematic collection of runtime information through observation of software systems and the conception of modeling approaches to

understand and improve performance properties. Software analytics provides actionable insights to developers from data generated or extracted from software systems [Menzies and Zimmermann, 2013]. This thesis lies on the intersection of software analytics and performance engineering. It introduces *Software Runtime Analytics for Developers*, which explores novel ways to extend developer's mental models by quantitative runtime aspects of their code by integrating data from production runtimes (specifically, runtime performance) into the software development workflow to present actionable insights. Existing mental models, informed by control- and data-flow, are thereby complemented with data from production runtimes that results in a more conclusive model of the program and enables the basis for data-driven decision making of code changes.

While the framework developed in the course of this thesis is general in the sense that it supports the integration of any kind of runtime information, we specifically focus our efforts to develop prototypes for *performance issues at production runtime* (which is reflected in the thesis statement).

> ***Thesis Statement:*** *"Software developers are enabled to identify and prevent significantly more runtime performance issues during software maintenance and debugging tasks when source code is augmented with information observed and collected at production runtime."*

To guide the design process throughout this thesis, we formulate two goals based on our hypothesis and thesis statement. We want to provide **operational awareness** to developers by extending their mental models with runtime dimensions. This should be achieved by tightly integrating data into the development workflow. By that, developers should become more aware of the operational footprint of their source code. At the same time, we also want to provide **contextualization** of these operational aspects. Runtime information should be available in the context of current development task to avoid context switches to other external tools (e.g., dashboards).

1.1 Research Questions

The objective of my research is to understand how software developers deal with challenges of runtime issues and to enable data-driven decision making based on data from production environments during development of software. To gain a better understanding of the problem domain and properly investigate the thesis statement, the following research questions are being investigated. An overview of the relationship between the questions is given in Figure 1.1. To answer these research questions, we conduct empirical studies and develop approaches as prototype implementations. Approaches are evaluated through case studies and controlled experiments.

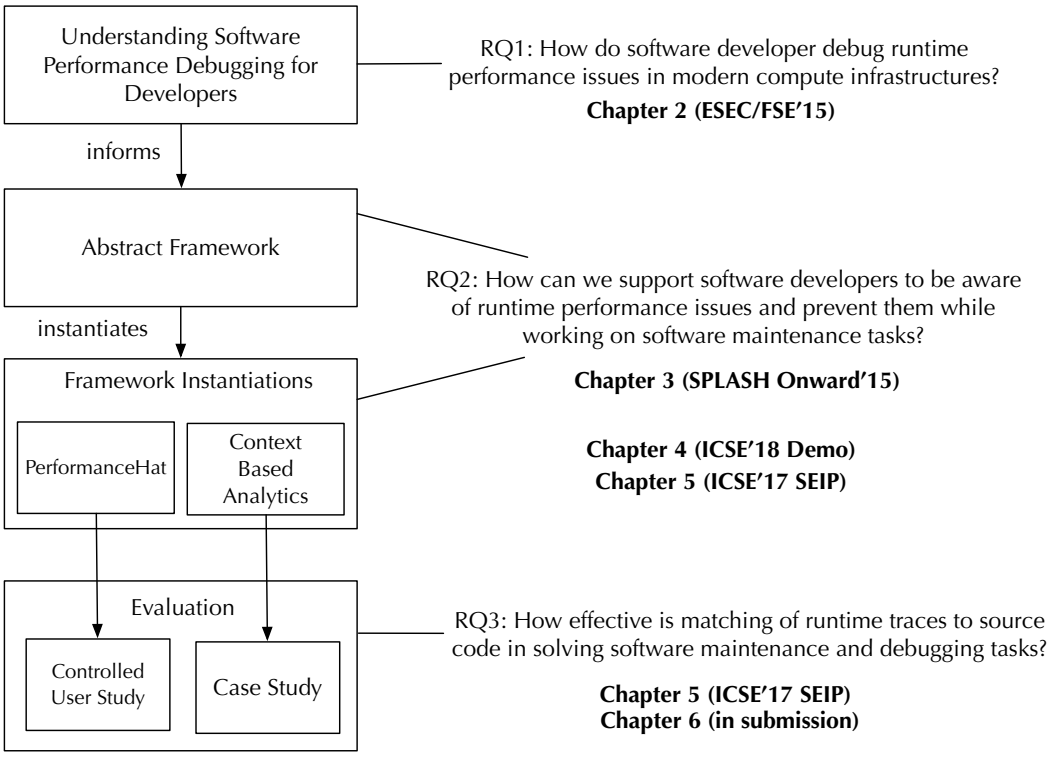


Figure 1.1: Overview of Research Questions

Research Question 1 (RQ1) – Understanding Software Performance Engineering for Developers: *How do software developer reason about runtime performance issues in cloud computing environments?*

The first research question serves as a basis for understanding the process of software performance engineering, specifically from the perspective of developers that deploy in cloud computing environments. The findings relevant for this thesis are part of a larger empirical study about software development practices in the cloud, that informed the design of the subsequent research questions. We conducted a mixed-method empirical study, consisting of semi-structured interviews with 25 software developers in 2 phases and survey with 294 participants. We used a mixed methodology consisting of three steps of data collection and iterative phases of data analysis [Bratthall and Jorgensen, 2002]. First, we defined a set of purposefully open-ended questions guided by our research question and conducted *qualitative interviews* with 16 participants. Second, to further substantiate the findings, we conducted a *quantitative survey* and gathered responses from 294 professional software developers. Using open coding, we identified 4 topics of high interest (one of them being how developers deal with runtime and performance issues). To gain a better understanding and more details on these topics, we then conducted a second round of *qualitative deep-dive interviews* with 9 professional developers.

In the context of software performance engineering, we present results around developer's perception of runtime information, their reactive approach to debugging, and their tool usage. We observed that, while cloud application developers are aware of the existence of runtime information, the current mode of presentation (i.e., external dashboards) makes it hard for them to gain actionable insights to make data-driven decisions during software maintenance and debugging tasks. We hypothesized that tighter integration of runtime information to source code can potentially enable awareness of operational aspects of code.

Research Question 2 (RQ2) – Software Runtime Analytics for Developers: *How can we provide awareness of runtime performance issues for developers and prevent these issues while working on software maintenance tasks?*

Informed by the findings of Research Question 1, we design an approach to integrate runtime information with source code to create awareness of runtime aspects in the development workflow. We answer this research question in two parts:

Abstract Framework: We describe a framework that is an abstract representation of potential solution spaces that can guide implementations to answer our research question. The framework consists of the following elements:

- *Specification:* Binary relation between code elements and runtime traces, represented as queries parameterized by element type.
- *Trace Filtering:* An extension to specification that allows for restriction of the trace space based on its attributes (e.g., filtering for regions, device types, sessions).
- *Impact Analysis:* Establishing a relation between changes in code to elements of the program that are impacted.
- *Presentation:* While the previous elements handle the relationship between abstract structures, these have to be presented and visualized to the developer in an actionable way.

Framework Instantiations: We describe the implementation of our two prototypes as instantiations of our framework. Both of our prototypes implement the framework around the idea of connecting runtime traces with source code elements, but support two orthogonal use cases. The prototype *PerformanceHat* is embedded in the IDE to create operational awareness for software maintenance tasks and provide early warning of potential runtime issues. The prototype *Context-based Analytics* supports

the reactive debugging process by establishing a graph structure where nodes represent both runtime traces and source elements, and weighted edges represent the degree of their relationship.

Both framework and prototypes were developed iteratively and benefited from reciprocal feedback loops from each other. They were refined interactively in every iteration that also included feedback from pilot case studies. Each of the prototypes covers a different use case of integration into the workflow. *PerformanceHat* demonstrates the case of integration in software maintenance to prevent runtime issues before deployment. *Context-based Analytics* demonstrates the case of debugging runtime issues by exploring the solution space through relations to source code.

Research Question 3 (RQ3) – Evaluation *How effective is matching of runtime traces to source code in solving software maintenance and debugging tasks?*

To evaluate our approach, we assess the its effectiveness by conducting empirical studies. We use our developed prototypes as a basis for the evaluation to perform a *controlled user study* and a *case study*:

- *Controlled User Study for PerformanceHat:* We conducted a controlled experiment with 20 professional software developers, in which they worked on software maintenance tasks that are designed to introduce runtime performance problems. Study participants are assigned into a treatment- or control group, where they either work with *PerformanceHat* or a representative baseline (*Kibana Dashboard*[1]). We found that, using our approach, developers are significantly faster in detecting and finding the root-cause of performance problems.
- *Case Study at IBM for Context-based Analytics:* In collaboration with IBM, we conducted a case study of *Context-based Analytics* on two common scenarios for runtime problems using traces from the company's active-deploy service within IBM Bluemix. The

[1] https://www.elastic.co/products/kibana

tool allows developers to release new versions of their cloud-based software with no downtime. We compare the number of steps taken and number of traces inspected in contrast to a tool commonly used by IBM engineers for runtime problem diagnosis, the *Kibana dashboard*. We found significant reductions in both investigated metrics.

1.2 Findings

In the following, we briefly summarize the findings of our efforts with respect to the presented research questions in Section 1.1. The foundation of these findings is a collection of selected publications that were published in international, peer-reviewed venues in software engineering research. The complete studies are presented in Chapters 2-6.

1.2.1 Understanding SPE for Developers (RQ1)

Software Performance Engineering (SPE) spans a broad range of activities in the software engineering life-cycle that relate to meeting performance objectives [Smith, 1993, Woodside et al., 2007]. While our exploratory study investigated a broader goal of characterizing software development in the cloud, we present the findings that guided the work of this thesis in the context of SPE. Specifically, we set out to understand the developer's perspective on how to leverage information gathered at runtime for debugging and maintenance activities when performance problems are reported in production of cloud computing systems.

Developer's Perception of Runtime Traces and Tool Usage

We were interested in how developers use runtime traces in their work to guide decisions about how to change their code. Through analysis of our initial interviews and the open questions in our survey, we identified that *runtime performance metrics* are of high interest to developers of our study. 62% of

survey respondents agreed that they have more runtime metrics available than before moving to cloud systems. System and application performance metrics were deemed to be the most interesting by our survey participants, where 84% state that they look at performance metrics on a regular basis.

In the survey, we also asked what tools developers specifically use in development for the cloud, which they have not used before moving their deployments to the cloud. 124 people responded to the question with multiple tools, which we categorized and quantified (more detail on our particular methodology in Chapter 2). The top two categories are tools that deal with understanding phenomena that occur in production. Performance management tooling was mentioned in 57% of cases. Log analytics tools was mentioned in 39% of all entries. This gives an indication of the importance of runtime information in software development for the cloud. We substantiate our initial findings through further interviews.

Developer's Reactive Approach to Performance Debugging

In deep-dive interviews, we investigated how our study participants approach a particular runtime performance issue by using information extracted or observed from production systems to solve the problem. We were able to identify a common thread in the way our interview participants approach these issues. The first common theme is that traces and metrics are usually not defined by the developer, but by someone who is concerned with running the application in production (i.e., operations engineers[2]). Traces and metrics were then all provided in dashboards accessible to everyone in the team. Operations engineers were the ones that actively observe the data on a more regular basis. Developers followed a more reactive approach, i.e., they acted on alerts or reports on tools like issue tracker. However, when actively debugging a known performance issue, all interview participants would first go *"by intuition"* and attempt to reproduce the issue in their own development environment, before inspecting how traces and metrics have evolved in production in the provided dashboards. Only if the local inspection did not yield any results, they would dig deeper into information

[2]Depending on who we were talking to, this role was also called DevOps engineer or system administrator

from production. Interviewees stated that they choose to forgo the data at first, because of the intricacies of navigating performance dashboards, while at the same time navigating through code. To quote one of our participants: *"I try to reproduce and solve the issue locally. Looking for the particular issue in the dashboard and jumping back and forth between the code is rather tedious".*

Implications for Software Performance Engineering

When solving problems that have occurred at runtime, they rather rely on beliefs and intuition than utilize metrics. This has also been observed for other software engineering activities in general [Devanbu et al., 2016]. The results of our study indicate that software developers are struggling to incorporate runtime performance information in their daily workflow to guide their software maintenance activities.

Data collected on application services in production is usually sent to monitoring and management services, and surfaced in a way that helps to make *operating decisions*. In our study, we observed that the required data to support maintenance and debugging activities is not made available in a manner that supports making *software development decisions*. To support software performance engineering for developers, we require new approaches to integrate operational aspects from production into the daily worklow of software developers.

1.2.2 Software Runtime Analytics for Developers (RQ2)

Findings from *Research Question 1* (see Section 1.2.1) indicate that to support developers in making data-driven decisions about runtime aspects when writing code, we need software analytics that are specifically targeted for software developers and integrated into their daily workflow.

We answer this research question in two parts:

1. We present an abstract framework that consists of the essential elements and their interrelations that capture all aspects relevant to integrating runtime information into the daily workflow of software developers to support data-driven decision making about code changes.

2. We present two different instantiations of this framework. *PerformanceHat* is an IDE integration to create awareness about runtime performance issues and potentially prevent them through light-weight performance impact analysis. *Context-based Analytics* is a web-based tool to support debugging of runtime issues by establishing explicit links between traces and source code. These prototype implementations act as proof-of-concepts and form the basis for evaluating the effectiveness of the approach.

While the abstract framework is designed to theoretically capture any kind of (runtime) information in the process, our implemented prototypes very heavily focus on software performance and reliability aspects at runtime.

Abstract Framework

The basic theory underlying our framework is that static information available while working on maintenance tasks in software development does not suffice to properly diagnose performance problems. The conjecture then is that augmenting source code with dynamic information from production runtime complements developer's mental models such that they can make informed, data-driven decisions on how to perform code changes. We propose a framework that provides the abstractions necessary to integrate runtime traces to source code in the development workflow. We depict these abstractions and their relationships in Figure 5.1 and summarize them in the following. The components of the framework should be seen as guidance to instantiate effective tools for software runtime analytics.

Mapping/Specification. In an initial step, we parse source code to an abstract representation, its Abstract Syntax Tree (AST). Nodes of the AST are then combined with applicable runtime traces in a process called specification or mapping. A node in the AST can be distinguished by its type (e.g., method invocations, collections, loop headers). Based on this type, we can define *specification queries* which determine how a set of traces can be mapped to elements of the AST based on criteria in both node and trace. This criteria is called a *specification predicate*.

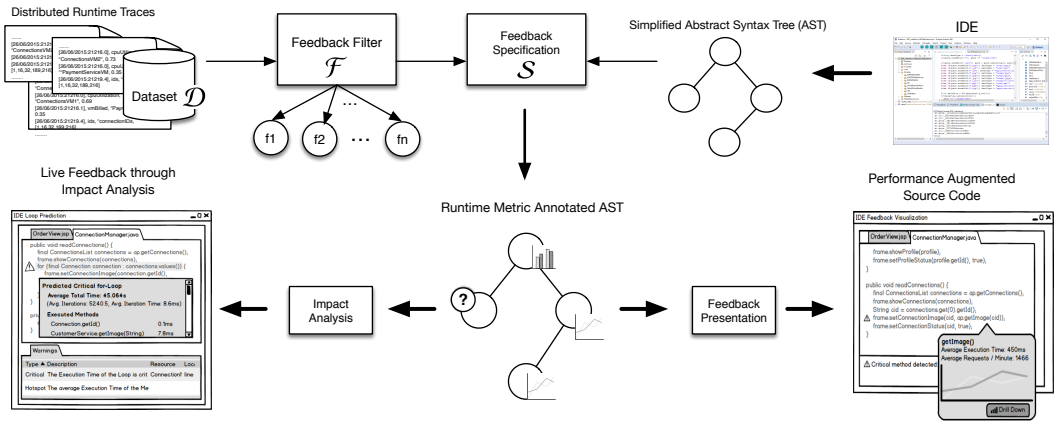


Figure 1.2: Framework for the software runtime analytics approach. Nodes of source code, represented as an Abstract Syntax Tree (AST), are augmented with runtime information. Results of the analysis are presented in the development workflow.

The predicate decides based on an expression over attributes within the source code node and runtime trace whether there is a match. While a specification query can take any form as long as it maps a set of trace data points to an AST node, we illustratively present two possible instances:

- *Entity-Level Specification:* In its simplest form the predicate returns true if information of one trace can be exactly mapped to one source code entity based on an exact attribute matching. This is a common use case for execution times or method-level counters that can be mapped to method definitions and method invocations.

- *Approximative System-Level Specification:* A more complex query could attempt to approximately map system-level traces to nodes. This approach would sample system level runtime properties over time and at the same time, through instrumentation, sample entry and exit points of each methods in the program execution. Overlaying both measurements leads to an approximative measure of the impact of certain method invocations to the system level property. A concrete example of a metric where this kind of

approximate mapping can make sense is memory consumption, which is usually measured at the system or process level.

Trace Filtering. In modern cloud applications, runtime traces are observed and collected from scalable, distributed systems. Especially performance information observed in this setting often exhibits a multi-modal distribution. Attaching all data corresponding to a source code element might be misleading, as interesting phenomena of the underlying code might be hidden in aggregation. Trace filtering enables different use case scenarios by allowing for sensitivity analysis. By applying filtering criteria on potentially problematic trace distributions, developers can reason about the interaction of their code and the underlying process creating the problems. Additionally, certain trace data sets might require filters to remove measurement errors or any other kind of outliers (i.e., data cleaning). This can be achieved either by removing numerical errors through fixed thresholds to more complex filtering with interquartile ranges or dynamic filtering adjusted based on trends and seasonality (in the case of time series).

In conclusion, trace filtering in the process of software runtime analytics enables the interplay of exploratory data analysis in conjunction with code analysis.

Impact Analysis. Developers regularly change code during software maintenance tasks. Impact analysis on source code level relates changes in code (change sets) to elements of the program that are affected by the change (impact set) [Lehnert, 2011a]. For our work, we consider impact analysis that is *predictive* (i.e., estimating future states of the program based on code changes) and *descriptive* (i.e., finding relations from impact set to their originating code change). For both kinds of impact analysis, the prerequisite is the mapping of runtime information to source code elements through specification queries.

In the predictive context, impact analysis is concerned with providing early feedback on how these changes would affect runtime properties of the software in production. The elements of predictive impact analysis are *inference* and *propagation*. After existing traces are mapped to code elements in the specifica-

tion phase, we infer properties of new elements with light-weight performance inference models to ensure timely feedback. Given inferred information on new code elements, update the information of all dependent code elements. This early feedback should enable developers to prevent performance issues from being deployed to production. From the perspective of the code's abstract representation, changing code means adding and deleting nodes from the AST. A prediction model is given the delta between the annotated AST, with all its attached runtime traces, and the current AST that includes the new changes, as parameters to infer information about potential future states about the unknown nodes in the current AST. The complexity of inference models can vary greatly, depending on the application nature. In the cases we present throughout this thesis (web applications deployed in the cloud), we use hybrid models, a mix of analytical and learning models [Didona et al., 2015, Desnoyers et al., 2012, Thereska and Ganger, 2008]. Analytical models capture properties of the programing- and execution model (e.g., execution times in single-threaded imperative programming can be computed through an additive model). Learning models capture dynamic effects of the production runtime that cannot be easily formalized in analytical models (e.g., scaling effects in elastic cloud computing systems).

In the descriptive context, impact analysis is concerned with introducing structure from an impact set to their originating source code change. It establishes explicit relationships from occurred runtime issues (manifested as traces) to source code or other parts of the program (e.g., configuration code or other traces) that need to be analyzed to diagnose the issues. Navigating from traces that had an undesirable effect on the system to other information sets takes the form of reachability analysis [LaToza et al., 2007].

Presentation. Results of the previously described analysis steps need to be surfaced to developers to reason about and, potentially, interact with (e.g., allow for trace filtering through an interface).

At the presentation stage, we are usually given a set of data points (distribution), rather than just a point estimate to present to developers. The type of visualization that is appropriate and is actionable for developers very

much depends on the use case of the concrete instantiation of the framework. A possible way to present runtime performance data could be to, for instance, display a summary statistic or simple visualization as initial information that is attached to an element of interest (e.g., a source code element) and allow for more interaction with the data in a more detailed view.

One of the studies in this thesis integrates the analysis into the Integrated Development Environment (IDE). In an IDE, the presentation process could result in an augmented source code view, that allows developers to examine their code artifacts annotated with runtime traces from production environments (e.g., method calls with execution times, usage statistics for features, or collections with size distribution). A similar kind of presentation could be placed at a different stage of the workflow, for instance, at the code review stage.

Framework Instantiations

We built prototypes that serve as proof-of-concept implementations based on the abstract framework. In the following, we briefly summarize the approaches *PerformanceHat* and *Context-based Analytics*. Table 1.1 provides an overview of how elements of the abstract framework map in the two prototypes.

Table 1.1: Overview of how the implemented prototypes have instantiated elements of the abstract framework

	PerformanceHat	**Context-based Analytics**
Specification	Identity matching on fully qualified method name in Java (Entity Specification)	Probabilistic matching based on semantic similarity (Approximative Specification)
Trace Filtering	Database views in local trace storage	Adjustment of time series windows in interface
Impact Analysis	Light-weight performance inference models for adding method invocations and loops into existing method contexts	Exploring impact set through transitive relations from traces to source code
Presentation	In-situ visualization in Eclipse IDE on methods and loops	Graph visualization of explicit links between source code and traces

PerformanceHat. We implemented *PerformanceHat*, an Eclipse IDE plugin for Java programs and a local server component with a database (dealing with storing a local model of trace information and filtering). Performance traces (e.g., execution times and CPU utilization) are attached to method definition and method calls. Specification from nodes to trace data is done through identity matching on method signatures. Source code elements that have information attached are highlighted in the IDE. Hovering over these elements reveals a tooltip with more information: For nodes with existing information, it shows a point-estimate of the execution time of the method. For inferred nodes, it shows the prediction in the form of a point-estimate, but additionally lists all parameters that lead to the prediction.

Our analysis is hooked into Eclipse's incremental builder to achieve tight integration into the development workflow. This means that program analysis, specification/mapping, and inference are triggered every time a file is saved and are applied to new and modified code. Impact analysis implements inference models for adding method calls and loops within a method body. Figure 1.3 provides a screenshot of the frontend in Eclipse. In this particular example, the developer introduces a new for-loop into an existing method. Impact analysis is performed and performance information for the for-loop is inferred immediately, showing the developer a prediction for the loop and the parameters given to the model. Given this information, the developer can now reason about the situation with data from the actual production runtime and make an informed decisions to act accordingly (e.g., by introducing a cache).

The complete tool chain of PerformanceHat included extensive setup documentation is available as an open source project on GitHub[3].

Context-based Analytics. We implemented *Context-based Analytics*, a prototype in Python, that, given a model of runtime traces in the system, establishes explicit links between traces and source code fragments. All link information is organized as a graph structure. Each node in the graph corresponds to a *fragment* (e.g., individual runtime traces, metrics as time series over a window, or

[3] `http://sealuzh.github.io/PerformanceHat/`

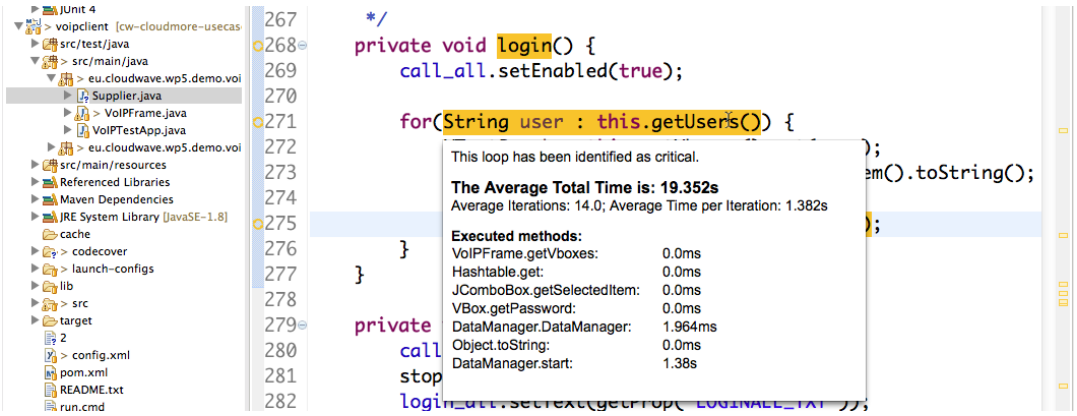


Figure 1.3: Eclipse plugin *PerformanceHat* applied on Java code from a VoIP application. The tooltip displays the information of a prediction that was triggered after a loop was introduced to an existing method.

source-code fragments). Edges between nodes correspond to a semantic similarity relation that link a fragment to related fragments, which we refer to as its *context*. Figure 1.4 illustrates the visualization of the graph in our implementation. Developers navigate the graph to inspect relevant connected fragments. Node types and its abstract relations (*specification queries*) are defined in an initial modeling phase. The graph is then constructed on-line from the modeled data sources.

Instead of querying and combing through results of many different tools to construct a mental model of the relationship between traces at runtime, developers would simply navigate the problem search space through clicking on their traces of interest and automatically see source code fragments that are related.

1.2.3 Evaluation (RQ3)

To evaluate whether the proposed approach has a significant impact on decisions made in the course of software maintenance or debugging tasks that impact performance, we conduct a controlled user study and a case study at IBM.

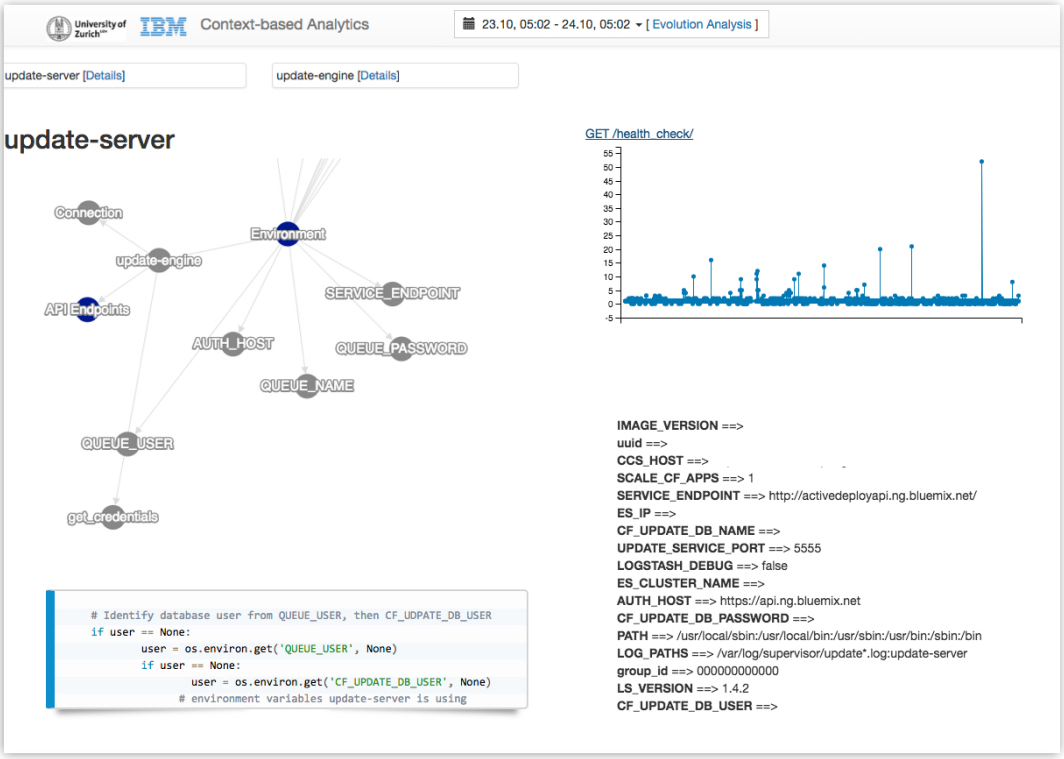


Figure 1.4: Context-based Analytics prototype implementation applied on an application within IBM. On the left the context graph and around it visualizations of different runtime traces and a connected code fragment. On the top right developers can adjust parameters for filtering

Controlled User Study

We evaluate our prototype *PerformanceHat* by conducting a controlled experiment with 20 professional software developers as study participants. We study whether our appraoch has a significant impact on decisions made in the course of software maintenance tasks, specifically related to performance. The study uses a between-subject design with two groups. A control group that is provided with Eclipse and a common tool to display runtime performance information, Kibana, and a treatment group using *PerformanceHat* in Eclipse.

Maintenance Tasks & Hypotheses. As the subject application of our study, we use Agilefant[4], an industrial Java application (around 115 kLOC) that serves as our experiment subject. The vendor also provides the software as an open source project on Github[5], making it available for our study.

We present the study participants with maintenance tasks that are performance sensitive, but also with tasks that are not relevant to performance. The mixing of task types has two reasons. First, we want to see to what extent augmenting source code with additional information is a distraction in performance irrelevant tasks. Second, we want to avoid learning effects after initial tasks in study participants (i.e., them knowing that looking at performance data is usually part of the solution).

To guide our study, we formulate the following hypothesis:

- H_1**: First Encounter**
  Hypothesis: Given a maintenance task that would introduce a performance problem, software developers using our approach are *faster in detecting* the performance problem

- H_2**: Root-Cause Analysis**
  Hypothesis: Given a maintenance task that would introduce a performance problem, software engineers using our approach are *faster in finding the root cause* of the performance problem

- H_3**: Non-Performance Relevant Tasks**
  Hypothesis: Given a maintenance task that is *not relevant to performance*, software engineers using our approach are *not slower than the control group in solving the task*

[4]https://www.agilefant.com/
[5]https://github.com/Agilefant/agilefant/

For every task in our study, we measure the total time required to solve the task. For performance relevant tasks, we additionally measure the time until the *first encounter* (FE metric) of a performance problem and the time until the *root-cause* of the problem was detected (RCA metric). Our participants work on four different tasks (*T1–T4*), two of which are relevant to performance (*T2* and *T4*).

User Study Results. We perform a Mann Whitney U test, a non-parametric statistical test, to check for significant differences between the population of the two groups and Cliff's delta to estimate the effect size. Looking at performance relevant tasks, the treatment group performs better (in absolute terms) for all measurements. For both total times, the difference is significant (p-value < 0.05, Effect Sizes/Cliff's delta: 0.92 and 0.67).

For the FE metric (First Encounter), we see a significant difference in T2/FE (Effect Size/Cliff's delta: 0.92). In T4/FE, however, the difference is not significant. A possible explanation for this difference is inherently encoded in the structure of the task T4. In T4, the change to be introduced by the study participant already occurs in two nested loops. So, even without direct presence of any runtime information in the process, a software developer can reason that introducing yet another loop leads to an $\mathcal{O}(n^3)$ time complexity. In T2, however, the introduced performance problem was not as obvious by simply inspecting code without consulting runtime information.

For the metric RCA (Root Cause Analysis), both T2/RCA and T4/RCA show significant differences (Effect Sizes/Cliff's delta: 0.68 and 0.92) between treatment and control group. Even in the case of T4, where the first encounter was more easily detectable through static code inspection alone, the analysis required querying runtime information to pinpoint the root cause of the performance problem.

For both regular (non-performance) tasks, T1 and T3, we were not able to reject the null-hypothesis. Note that, this only means not enough evidence is available to suggest the null-hypothesis is false at the given confidence level. Thus, there is a strong indication that there are no significant differences for these tasks.

This is an indication that our approach does not introduce significant cognitive overhead that “distracts" developers in the context of regular, non-performance relevant, tasks.

Case Study at IBM

We evaluate the prototype *Context-based Analytics* in a case study with the company IBM. We conducted the evaluation at a company because it provided us with an environment to facilitate a case study in the field. The goal of this study is to evaluate to what extent our approach can be used to diagnose runtime issues in a real-life application. For that, we measure how much effort is required to diagnose a problem using our approach as compared to a baseline approach used within the company. The application used in the case study is IBM's active-deploy[6], which is a utility application that allows the release of a new version of an application with no down time. It is implemented as a service-based architecture and consists of approximately 84 kLOC. Runtime information is gathered in the form of system metrics and log messages. System metrics are observed through a system crawler.

Designing studies always involves trade-offs. We chose to do a more in-depth case study in one company, rather than aiming for larger coverage of companies with potentially shallower case studies.

Scenarios & Metrics. To study our approach based on realistic situations, we must evaluate them for runtime issues that are representative of common real-life situations. Thus, we extract two scenarios of runtime problems in active-deploy with roots in software development that are considered “typical” by application experts from IBM. We divide these scenarios into 8 concrete subtasks (i.e., questions that an engineer diagnosing these problems needs to answer), which we have designed in collaboration with the same application experts. We aim to establish a proxy for the effort that engineers go through to diagnose the two scenarios, including all 8 subtasks, by introducing two metrics:

[6]https://github.com/IBM-Bluemix/active-deploy

1. *Number of steps taken* counts all distinguishable activities taken that might yield necessary information to be able to reach a conclusion of the given task.

2. *Number of traces inspected* counts all information entities that have to be investigated to either solve the task or provide information that guide the next steps to be taken.

Case Study Results. The results of this case study evaluation show that, aggregated over all 8 tasks combined, our approach saved 48% of steps that needed to be taken and 40% of traces that needed to be inspected. There is only a single subtask, *Task 4*, where using our approach leads to an increase in the number of traces that need to be inspected. For 3 subtasks (*Tasks 3, 6, and 7*), the number of inspected traces is unchanged. However, the number of steps taken increases substantially for all subtasks. Not that, in general, the case study design is chosen to present a conservative measure of saved effort. Specific domain and data model knowledge is often necessary to construct proper queries to the data sources available in the baseline. Particularly, we represent an idealized situation in our comparison in which the search in the baseline tool does not contain any unnecessary diagnosis steps. That means, in our case study scenarios, developers never run into a “dead end" while diagnosing the issues. Our approach already encodes the domain model in the specification queries. This makes it more attainable for developers with less experience within the domain and in general.

1.3 Limitations and Scope of Work

The framework presented for the proposed approach is modeled on a fairly high abstraction level. The concepts can work with any programming language that produces execution traces (that can be indexed and are available during design time) and from which we can parse an abstract representation in the form of an AST. However, for practical reasons the scope of the work is limited in several dimensions:

- *Cloud Applications*: We target cloud applications that are delivered as a service (SaaS applications), which are accessed by customers as web applications over the Internet or are consumed by other services over an API. We specifically do not consider embedded systems, "big data" applications, scientific computing applications, or desktop applications. While the framework presented in this thesis could, in theory, also be used for these applications, we have no studies that would show feasibility or effectiveness outside the described scope.

- *Object Oriented Programming:* The current prototype implementations of the proposed approach abstracts the components of the framework and provides proper extension points. Due to details in the implementation, however, the implemented approach currently only works with object-oriented programming languages (Java for *PerformanceHat* and Python for *Context-based Analytics*). However, the concepts can be easily transferred to other programming languages and paradigms (as long as they have observable units, see below).

- *Observable Production Systems:* We require a system that exposes enough data about itself at runtime. It is explicitly not in our scope to contribute new instrumentation methods, but rather making the already exposed data actionable for developers. Further, this means that our approach is limited to modeling only methods that are actively measurable in production by existing APM tools (e.g., DynaTrace, Kieker). Therefore, methods that might have suboptimal (theoretical) computational complexity, but do not exhibit any significant overhead that can be observed (e.g., due to low input sizes), will inherently not be part of our analysis.

- *Software Maintenance:* Another limitation is that the proposed approach only provides value when working on maintenance tasks. If there is no existing data observed at production runtime that can be used for mapping or prediction, the approach cannot help.

- *Prediction Model Quality:* Impact analysis is only as good as the provided prediction model. However, it is not in the scope of this thesis to come up with a perfect performance model for any application type, but rather to investigate whether augmentation of additional performance information has an impact on solving performance maintenance tasks.

1.4 Opportunities and Future Work

We have demonstrated that when source code is augmented with runtime information and is integrated into the development workflow, software developers are able to identify, prevent, and diagnose performance problems faster. These findings are promising and provide the foundation for interesting opportunities and future work.

Model Interpretability. We aim for fast feedback loops that can help make data-driven decisions about code changes based on data of the operational aspects of code elements. Because we integrated inference in the incremental build process in *PerformanceHat*, we required models that provide immediate results. Previous work has shown that there is a trade-off between model accuracy and speed (i.e., time-to-prediction) [Jiang et al., 2012]. We argue that a lack of accuracy in the prediction can be mitigated with interpretable models. Providing information about what exactly makes a model arrive to a certain prediction gives developers another dimension to reason about the instant feedback. We did initial work in this area by exposing input parameters to our models to developers, so that they can understand what particular values led to a certain prediction. Future work can expand on these initial efforts by, for instance, allowing for sensitivity analysis (i.e., allowing to change input parameters to model on the fly).

Exploring Different Runtime Models. While the abstract framework is designed to be general in the data it models, the prototypes and evaluations had a focus on performance aspects. It would be interesting to investigate other dimensions of runtime models that require more complex and approximative

specification functions. We have initial work in this area that integrates costs of cloud instances into the source code [Leitner and Cito, 2016]. Other examples that stand to reason are memory consumption, energy efficiency, or distribution of exceptions.

(Crowdsourced) Benchmarking Results. While we focus on integrating data from production environments, we could just as well integrate runtime information resulting from performance benchmarks in any kind of environment (e.g., staging). Another avenue of work that would be interesting is to establish a crowdsourced database of benchmarking results from open source systems. We imagine that performance benchmarks, that are defined in open source repositories (which is the case for many Go projects on GitHub, for instance), would be continuously executed on different hardware with potentially different configuration parameters. Results of these benchmark executions would be populated to a community-driven metrics database. When integrating this information in source code, we can make a probabilistic argument about performance based on these previously recorded executions.

Integration to Code Review. In our work, we integrated operational aspects into developer's code view to prevent runtime issues already in development. However, the approach is more generally applicable to any kind of source code view. Future work could study whether applying our methodology and analysis during code review is a stage in the development process that can benefit from our approach.

1.5 Related Work

This thesis combines aspects of many broad topics in software engineering, systems, and visualization. We briefly review work related to our research that can be broadly categorized in the following categories: *software analytics*, *understanding runtime behavior*, and *impact analysis*. Chapters 2 to 6 of this

thesis additionally explicate related work that is specific to the research elaborated in the respective chapter.

1.5.1 Software Analytics

There is a variety of artifacts that is generated in the development of software systems (e.g., software history in version control [Zimmermann et al., 2005], issue tracker data [Fischer et al., 2003], developer communication [Bacchelli et al., 2012]) and observerd or extracted during the execution of software systems (e.g., logs and telemetry [Barik et al., 2016], performance and reliability metrics [Han et al., 2012], usage data [El-Ramly and Stroulia,]). Software analytics is a broad research area that is concerned with the systematic collection, analysis, and modeling of these software artifacts with the goal of supporting data-driven decision making through actionable insights for stakeholders surrounding all aspects of creating software [Buse and Zimmermann, 2012, Menzies and Zimmermann, 2013, Zhang et al., 2013]. Menzies and Zimmerman give an overview of the broad areas of software artifacts that are being investigated in the context of software analytics [Menzies and Zimmermann, 2013]. Most of the work done in “classical" software analytics mines static information to extract actionable insights. Work in the area that is more closely related to our work is briefly summarized in the following. *Log Advisor* applies learning techniques from existing logging instances to provide guidance for developers on where to log in a code base [Zhu et al., 2015]. Stackmine mines callstack traces from online services at Microsoft to discover performance bugs causing high impact delays [Han et al., 2012]. Using real-world execution traces of device-drivers, Yu et al. identify patterns of runtime behaviors that are likely to cause previously measured impacts [Yu et al., 2014].

Our work is similar in the sense that we also use runtime information to construct models to understand runtime behavior and provide impact analysis. The focus of our work is to create a more tight integration of this data into source code to aid the programming and debugging process of developers and enable faster feedback loops.

1.5.2 Understanding Runtime Behavior

Visualization supporting system and program comprehension is a large field that covers many use cases. There have been several literature reviews that look into visualization of runtime aspects of software. Merino et al. published a survey that covers actionable visualizations in software development [Merino et al., 2016], in which they break down approaches by audience (e.g., developer, project manager, tester) and data source (e.g., source code, version control, running system). Isaacs et al. survey the state-of-the art of visualizations of various types of traces to aid software performance engineering activities [Isaacs et al., 2014].

Visualization in Code Views. Work that is more directly related to ours augments specific statements and expressions in source code with visualizations of runtime behavior. Cornelissen et al. introduces the visualization of traces through circular bundle views in the IDE and show that trace visualization can significantly improve program comprehension [Cornelissen et al., 2011]. Senseo [Röthlisberger et al., 2009] is an approach embedded IDE that augments the static code view perspective with dynamic metrics of objects in Java. Theseus [Lieber et al., 2014] augments JavaScript code in the debug view in the browser with runtime information on call count of functions asynchronous call trees to display how functions interact. The work closest to our implementation of *PerformanceHat* is by Beck et al., who augmented method definitions in the IDE with in-situ visualization containing information retrieved from a local profiler [Beck et al., 2013]. However, their approach is limited to local data for the visualization from sampling stack traces. Our work differs first in the underlying data source, which are traces from production systems, rather than from local profilers. We also provide the ability to introduce filtering mechanisms on the underlying data, enabling more debugging and comprehension use cases. Also, our approach goes beyond just augmenting information of existing traces, but attempts to predict the impact of new code through light-weight inference models.

1.5.3 Impact Analysis

Our work takes impact analysis techniques found in software engineering (specifically, program analysis) and combines them with impact analysis in the context of software performance. We briefly introduce related work in both subareas of *change impact analysis* and *performance modeling and prediction*.

Change Impact Analysis. Change impact analysis, in the context of software development, supports the comprehension, evaluation, and implementation of changes in software. Lehnert provides a comprehensive, in-depth review of software change impact analysis [Lehnert, 2011a]. Impact analysis can be seen from two different perspectives, that are reflected in the instantiations of our framework. It can either be seen as the process of estimating the global effect of local changes to a software system in terms of introduced or modified code (as in *PerformanceHat*). It can be also seen establishing a relation to other artifacts in the program that need to be modified to achieve a desired program state (as in *Context-based Analytics*). We focus on work that provides immediate impact analysis on source code level (as in [Do et al., 2017, Yazdanshenas and Moonen, 2012, Ren et al., 2004, Orso et al., 2004]). The process of impact analysis on code level is described as taking the change set of a program and computing an impact set. The change set consists of all newly introduced and modified code elements. The impact set contains elements of the program that are affected by the introduced change. Estimating the impact set generally takes the form of reachability analysis between elements of the program (i.e., propagation). Li et al. provide the most recent systematic literature review on code-based change impact analysis techniques [Li et al., 2013].

Performance Modeling and Prediction. The goal of impact analysis in our work on *PerformanceHat* is to provide immediate feedback of runtime performance properties of code level changes. Feedback is given in the form of execution time predictions on method level. To achieve that goal, it draws from work from performance modeling (encoding assumptions), and performance prediction (inferring properties based on these assumptions), and program analysis (extracting

model parameters and impact propagation). We are specifically interested in model-based approaches that are applied early in the development workflow to support various design and implementation decisions. Our work is orthogonal (and could be potentially combined) with research that attempts to estimate the impact of configuration changes on system performance [Jamshidi et al., 2017, Sarkar et al., 2015, Siegmund et al., 2015, Zhang et al., 2015].

In the context of this work, we broadly characterize these models as *analytical models*, *simulation models*, and *machine learning*. Simulation models are out of scope for our work, which integrates prediction into the incremental builder in the IDE, and thus requires fast models. We briefly address work in analytical and machine learning modeling for performance predictions. Analytical models explicitly represent the relation between performance output (e.g., execution time) with a vector of input parameters (e.g., workload, hardware) as a mathematical equation. Many existing approaches use analytical queuing models [Di Sanzo et al., 2012, Singh et al., 2013, Jiang et al., 2010, Casale et al., 2008] or petri nets [Brogi et al., 2007, King and Pooley, 2000, Bernardi et al., 2002] to represent system models. Machine learning is a black-box approach that learns parameters of performance models from existing traces (training data) that are seen as robust with respect to changes to the system [Venkataraman et al., 2016, Couceiro et al., 2010, Ozisikyilmaz et al., 2008]. For more in-depth information, we refer to the work of Balsamo et al., who provide a comprehensive literature review on model-based performance prediction in software development [Balsamo et al., 2004]. More recent work surveys design-time performance models [Brunnert et al., 2015], as well as performance models for cloud computing [Ardagna et al., 2014].

To allow for the kind of immediate impact analysis we aim for, we need fast models that provide almost instant feedback to developers. Our work is more in line with existing research on hybrid models combining analytical and machine learning models [Didona et al., 2015, Desnoyers et al., 2012, Thereska and Ganger, 2008]. Hybrid models augment analytical models (capturing system and program structure) with machine learning models (capturing dynamic effects of the system). These effects are difficult to reliably express in terms of analytical

formulas or as parameters. This can, for instance, be system internals that are not known and cannot be extracted (e.g., performance variation of cloud instances [Leitner and Cito, 2014]). We know that different use cases require different inference models. Thus, the abstract framework presented in Section 1.2.2 models impact analysis as an extension point (which is also manifested in our tool *PerformanceHat* through modularization). For instance, when using our framework in the context of real-time (embedded) systems, an inference model might take the form of worst-case execution time (WCET) analysis [Bernat et al., 2002].

1.6 Summary of Contributions

The research for this thesis was initiated to gain an understanding of a specific aspect of the software performance engineering process for developers with the goal of recognizing problems that can be addressed by research. The findings of this initial study informed the creation of an abstract framework that encompasses all aspects of incorporating information of systems at runtime, specifically performance, into the software development workflow by mapping traces to source code elements and enabling inference on code changes to facilitate fast feedback cycles. We implemented two prototypes as instantiations of this framework and evaluated their effectiveness through controlled quantitative experiments and case studies. The findings on the evaluation of *PerformanceHat* support our *Hypothesis 1* that developers can prevent runtime issues from reaching production when their mental models of a program are complemented with operational aspects of code elements. Further, *Hypothesis 2* finds support from both the studies on *Context-based Analytics* and *PerformanceHat*. Developers are faster in diagnosing runtime issues when an explicit link between source code element and runtime information exists.

Goals. The core of our work that is expressed through the elements of the framework has two succinct goals that guided the design process throughout the thesis. The achievement of these goals is part of the contribution of our work:

- **Operational Awareness:** By integrating runtime aspects into source code and, thus, into the development workflow, developers become more aware of the operational footprint of their source code.

- **Contextualization/Modularization:** Developers minimize context switches, since they do not need to specifically search for runtime information in external tools, but rather have them available in the context of the current task (i.e., module) they are working on.

Contributions. We summarize the contributions of this work as follows:

1. **Understanding Software Development in the Cloud:** We conducted a mixed-method empirical study that explored a wide range of practices including tools, data, and processes of software developers that deploy their code in the cloud. A particular aspect of findings in this larger study that observed software performance engineering practices from a developer's perspective served as the basis of understanding for the subsequent research questions of the thesis.

2. **Abstract Framework for Software Runtime Analytics:** We present an abstract framework that provides a comprehensive representation of the aspects relevant to integrating runtime traces into the daily workflow of software developers to support data-driven decision making, that is independent of concrete technologies and domains. It can be seen both as an explanation framework to better understand the aspects of developer targeted runtime analytics, as well as a structure that guides the development of concrete implementations.

3. **Prototype Implementations:** We present instantiations of our framework as prototype implementations. The implementations act as proof-of-concepts and form the basis for evaluating the effectiveness of the approach. Additionally, we provide open source versions of the prototypes for reproducibility.

4. **Empirical Evaluation of Effectiveness:** We present empirical evidence from a controlled experiment with professional software developers and case study conducted at IBM that, using our approach, software developers can identify and prevent significantly more performance problems during software maintenance tasks and debugging tasks when source code is enriched with runtime performance traces.

1.7 Thesis Roadmap

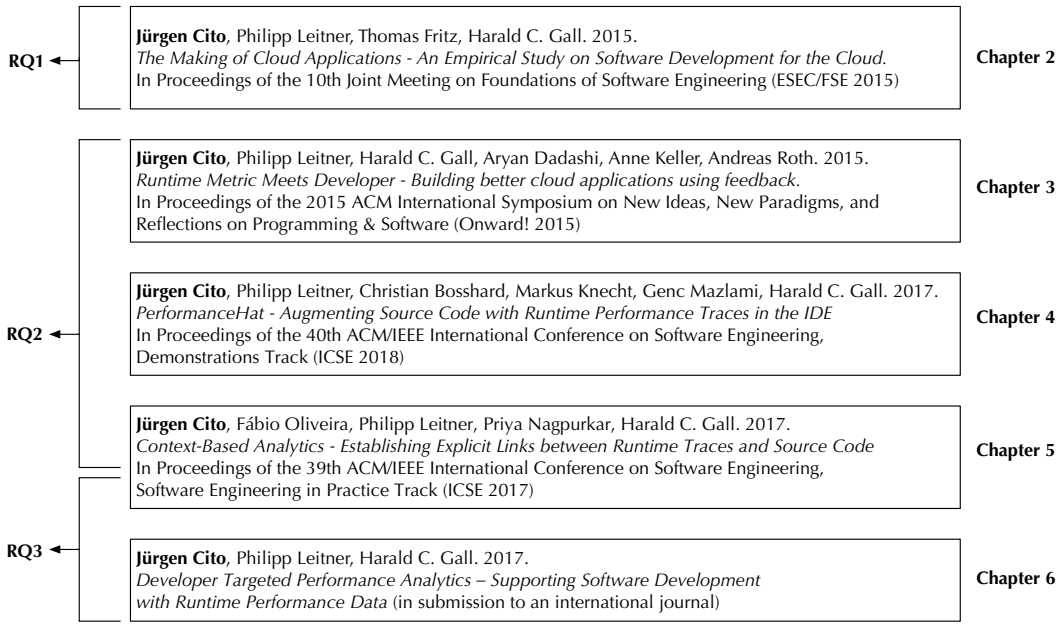


Figure 1.5: Roadmap of this thesis relating research questions to publications.

Figure 1.5 provides an overview of research activities organized by research question. The remaining Chapters 2 to 6 are each based on scientific publications in international, peer-reviewed venues. All publications were created in collaboration with my supervisors, Philipp Leitner and Harald C. Gall, and other

members of the University of Zurich. We indicate publications that were created with external collaborators accordingly.

Chapter 2 investigates broad aspects of software development for the cloud in a mixed-method empirical study.

Chapter 3 explores a framework to integrate runtime aspects of software into source code and presents different case studies. This publication was created in collaboration with SAP SE.

Chapter 4 presents design decisions and implementation of *PerformanceHat*, an instantiation of our framework that augments source code with performance traces in the IDE.

Chapter 5 introduces *Context-based Analytics*, an approach in which explicit links are created between source code fragments and runtime traces to form a graph structure. This publication was created in collaboration with IBM Research.

Chapter 6 presents an algorithm to efficiently match information to source code and reports on a controlled user study assessing the effectiveness of our approach on maintenance tasks relevant to software performance issues requiring performance traces from production.

Figure 1.6 lists publications that were part of my PhD, but are orthogonal to the core publications of this thesis. We broadly group them in *performance & cloud benchmarking*, *empirical studies on ecosystems & reproducibility*, and *program analysis & transformation*.

Figure 1.6: Further selected publications that were part of my PhD, but are outside of the scope of the thesis

2

An Empirical Study on Software Development for the Cloud

Jürgen Cito, *Philipp Leitner, Thomas Fritz, Harald C. Gall*
Published at the Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, 2015
Contribution: Planning and conducting interviews and survey, data analysis, paper writing

Abstract

Cloud computing is gaining more and more traction as a deployment and provisioning model for software. While a large body of research already covers how to optimally operate a cloud system, we still lack insights into how professional

software engineers actually use clouds, and how the cloud impacts development practices. This paper reports on the first systematic study on how software developers build applications for the cloud. We conducted a mixed-method study, consisting of qualitative interviews of 25 professional developers and a quantitative survey with 294 responses. Our results show that adopting the cloud has a profound impact throughout the software development process, as well as on how developers utilize tools and data in their daily work. Among other things, we found that (1) developers need better means to anticipate runtime problems and rigorously define metrics for improved fault localization and (2) the cloud offers an abundance of operational data, however, developers still often rely on their experience and intuition rather than utilizing metrics. From our findings, we extracted a set of guidelines for cloud development and identified challenges for researchers and tool vendors.

2.1 Introduction

Since its emergence, the cloud has been a rapidly growing area of interest [Buyya et al., 2009, Armbrust et al., 2010]. Several cloud platforms, such as Amazon's EC2, Microsoft Azure, Google's App Engine, or IBM's Bluemix, are already gaining mainstream adoption. Developing applications on top of cloud services is becoming common practice. Due to the cloud's flexible provisioning of resources, and the ease of offering services online for anyone, the cloud also influences software development practices. For instance, cloud development is often associated with the concept of "DevOps", which promotes the convergence of the development and operation of applications [Hüttermann, 2012].

There is currently significant research interest in how to efficiently manage cloud infrastructures, for instance in terms of energy efficiency [Beloglazov et al., 2012] or maximized server utilization [Marshall et al., 2011]. Another core area of interest in cloud computing research is its use for high-performance computing in lieu of an expensive computer grid [Iosup et al., 2011]. However, so far, there is little systematic research on the consumer side of cloud computing, i.e., how software developers actually develop applications in and for the cloud. Only

recently, Barker et al. voiced this concern in a position paper, stating that the academic community ought to conduct more "user-driven research" [Barker et al., 2014].

In this vein, this paper presents a systematic study on how professional software engineers develop applications on top of cloud infrastructures or platforms. We deliberately cover a broad scope, and analyze how applications are designed, built and deployed, as well as what technical tools are used for cloud development. We conducted a mixed-method study consisting of an initial interview study with 16 professional cloud developers, a quantitative survey with 294 respondents, and a second round of interviews with 9 additional professionals to dive deeper into some questions raised by the survey. All interview participants work at international companies of widely varying size (from small start-ups to large enterprises), and have diverse backgrounds with professional experience ranging from 3 to 23 years.

In particular, we address the following research questions:

***RQ 1**: How does the development and operation of applications change in a cloud environment?*

***RQ 2**: What kind of tools and data do developers utilize for building cloud software?*

Our research has important implications for cloud developers, researchers, and vendors of cloud-related tooling. Primarily, due to the volatility of cloud instances, developers need to accustom themselves to not being able to directly touch the running application any longer. That is, quick fixes of production configurations are equally impossible as logging into a server for debugging. As a research community, we need to investigate how to best support developers in this task, as well as analyze how code artefacts related to cloud instance management evolve. Finally, we have seen that more types of metrics get more and more important, but they are still not directly actionable for developers. Hence, we need to research better tooling that brings this data into the daily workflow of cloud developers.

The remainder of the paper is structured as follows. First, we provide some background on cloud computing terminology (Section 2.2), followed by a discussion of related work in Section 6.2. We present the study design in Section 2.4, followed by an in-depth summary of our findings (Section 2.5) and a discussion of the implications resulting from these findings (Section 2.6). We detail the major threats to validity of our research in Section 5.7, and conclude the paper in Section 3.8.

2.2 Background

While the term "cloud computing" is commonly ill-defined[1], the research community has widely gravitated towards the NIST definition [Mell and Grance, 2011]. As illustrated in Figure 2.1, this definition considers three levels, each defined by the responsibilities of IT operations provided by the cloud vendor. In an *Infrastructure-as-a-Service (IaaS)* cloud, resources (e.g, computing power, storage, networking) are acquired and released dynamically, typically in the form of virtual machines. IaaS customers do not need to operate physical servers, but are still required to administer their virtual servers, and manage installed software. *Platform-as-a-Service (PaaS)* clouds represent a higher level of abstraction and provide entire application runtimes as a service. The PaaS provider manages the hosting environment and customers only submit their application bundles. They typically do not have access to the physical or virtual servers that the applications are running on. Finally, in *Software-as-a-Service (SaaS)*, complete applications are provided as cloud services to end customers. The provider handles the entire stack, including the application. The client is only a user of the service.

PaaS clouds are particularly interesting for software engineers, as they allow them to solely focus on developing applications. They typically relieve the developer from having to care about any operations tasks, and handle varying system load transparently via auto-scaling. This ability to adapt to workload

[1]Oracle's CEO Larry Ellison once noted jokingly that he cannot think of a single thing Oracle does that is not "cloud".

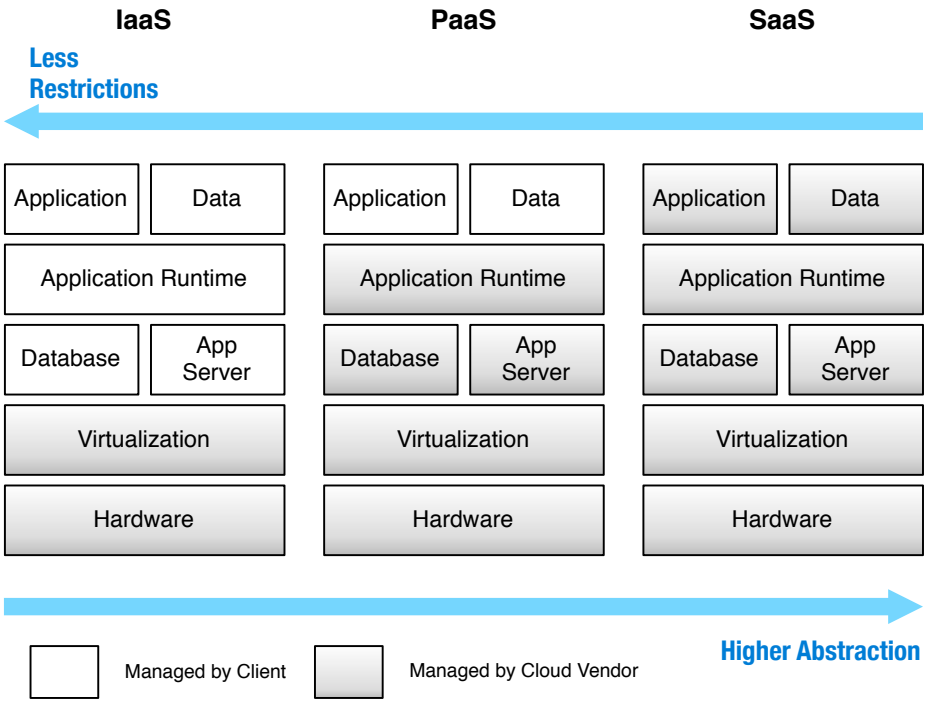


Figure 2.1: Basic models of cloud computing (following [Mell and Grance, 2011])

changes is referred to as *elasticity*. However, in order to do so, these platforms impose severe restrictions. For instance, they typically only support rather narrowly defined application models (e.g., three-tier Web applications), and require the developer to program against provided APIs. This often also leads to vendor lock-in [Lawton, 2008].

With IaaS, the idea of Infrastructure-as-Code (IaC) has also started to gain momentum. IaC allows users to define and provision operation environments in version-controlled source code. Essentially, in an IaC project, the entire runtime environment of the application (e.g., IaaS resources, required software packages, configuration) is defined using scripts, which can then be executed by tools such as Opscode Chef. These scripts allow entire test, staging or production environments to be started without manual interaction. The move towards IaC with its reproducible provisioning has become necessary since cloud

applications often consist of a large number of machines that have to be configured automatically to scale horizontally.

Another concept commonly associated with cloud development is DevOps [Hüttermann, 2012]. DevOps describes the convergence of the previously mostly separated tasks of developing an application, and its deployment and operation. In DevOps, software development and operation activities are often handled by the same team, or even by the same engineer. By aligning the goals of development and operations, DevOps aims at improving agility and cooperation.

2.3 Related Work

There has been a multitude of empirical research on the development of general software applications. For instance, Singer et al. have recently researched how developers use Twitter [Singer et al., 2014]. Murphy-Hill et al. have looked at how bugs are fixed [Murphy-Hill et al., 2013]. However, so far, very little empirical research has been conducted in the cloud computing domain, even though there are several calls for more research on software development for the cloud. Barker et al. [Barker et al., 2014] recently named "user-driven research" as one of the major opportunities for high-impact cloud research. Khajeh-Hosseini et al. [Khajeh-Hosseini et al., 2010] stated that the organizational and process-oriented changes implied by adopting the cloud is currently not sufficiently researched. While Mei et al. did not consider software engineering a major challenge for cloud computing in 2008 [Mei et al., 2008], they later on provided a whole list of software engineering issues to be tackled by research [Mei et al., 2009].

So far, research in cloud computing has mainly focused on provider-side issues (e.g., relating to server management [Beloglazov et al., 2012, Marshall et al., 2011] or performance measurement [Iosup et al., 2011]). Similarly, work by Bezemer et al. mainly focuses on performance problems on the server-side of SaaS applications, rather than development aspects of cloud software in general [Bezemer and Zaidman, 2014]. On the client side, some research has been conducted on concrete programming models. A large part of this research deals

with data analysis, typically using the Map/Reduce paradigm (e.g., [Palanisamy et al., 2011]). While interesting, these works do not cover the professional software development environment that we address with our study. Research on cloud programming models for non-scientific contexts is more limited. One example is the jCloudScale framework proposed in [Zabolotnyi et al., 2015] [Leitner et al., 2012]. jCloudScale is a Java-based middleware that aims to simplify the development of IaaS applications. A similar goal also motivated the research presented in [Jayaram, 2013], which investigated an extension of Java RMI for simplifying the development of elastic, cloud-based applications.

One aspect that is already reasonably well-understood in literature is how and when companies choose to adopt cloud computing, and for which reasons [Narasimhan and Nichols, 2011] [Gupta et al., 2013]. Both of these studies are concerned primarily with SaaS adoption. That is, they target cloud adoption by end users more than by professional software developers. This is not the case in a related industry study, dubbed the "DevOps Report" [dev, 2014]. This survey garnered over 9200 respondents, praising the DevOps idea as a key enabler of profitable and agile companies. Given that the source of this report is also a major player in the DevOps business, independent scientific evaluation to support these results would be valuable.

None of the work discussed so far has empirically evaluated how cloud software is actually developed in practice. The only work we are aware of that goes into this direction is a (not peer-reviewed) white paper on enterprise software development in the cloud [Shiver, 2014]. This report is based on a survey with 408 respondents. The report concludes that enterprise developers are largely not yet adopting the cloud, but if they do, they are able to improve time-to-market.

2.4 Research Method

To investigate how the cloud influences software development practices, we conducted a study based on techniques found in Grounded Theory [Hoda et al., 2012]. Following the recommendations in [Bratthall and Jorgensen, 2002], we used a mixed methodology consisting of three steps of data collection and iterative

Table 2.1: Method and Participants

Study Type	# Qs	Participants		Platform		Company		Experience
		#	IDs	PaaS	IaaS	large	small	Avg (± StdDev)
Interviews	**50**	**25**		**15**	**10**	**14**	**11**	**9 years (± 6.5)**
Interview_1	23	16	P1 - P16	13	3	12	4	9 years (± 7)
Interview_2	27	9	D1 - D9	3	6	2	7	8 years (± 6)
Survey	**23**	**294**		**103**	**191**	**102**	**192**	**9 years (± 5)**

phases of data analysis. First, we defined a set of open-ended questions from our research questions and conducted *qualitative interviews* with 16 participants. Second, to further substantiate our findings, we ran a *quantitative survey* and gathered responses from 294 professional software developers. Using open coding, we identified 4 topics of high interest. To gain a better understanding and more details on these topics, we then conducted a second round of *qualitative deep-dive interviews* with 9 professional developers. Table 2.1 provides a more detailed breakdown of our participants. All interview and survey materials can be found on our web site[2].

Qualitative Interview Study (Interview_1) Protocol. We conducted semi-structured interviews with software developers that had previously already deployed software in the cloud in a professional context. For these interviews, we defined a set of 23 questions based on our research questions. In the interviews, we covered all questions with each participant, but the concrete order of questions followed the "flow" of the interview. Interviews were conducted by the first author, either face-to-face on-site of the interviewee or via Skype. Interviews lasted between 30 and 60 minutes, were conducted in either German or English, and were audio-recorded.

Participants. Interview participants were recruited from industry partners and our personal network. To cover a broad range of cloud development experiences, we recruited participants from both, smaller companies (1 – 100 employees) and larger enterprises (> 100 employees). Participants had to either deploy on IaaS or PaaS as well as in public and private clouds. Furthermore,

[2]http://www.ifi.uzh.ch/seal/people/cito/cloud-developer-study.html

we also made sure to recruit some participants that delivered SaaS applications. Overall, we recruited 16 participants (P1 to P16), all male software developers with 3 to 23 years of professional development experience (average of 9 years ± 7 standard deviation), and from 4 different countries and two continents. Our 16 participants came from 5 different companies—12 participants worked in larger enterprises, 4 in smaller companies.

Analysis. After the interviews, we transcribed all audio recordings. The first two authors then used an open coding technique to iteratively code and categorize participants' statements, resulting in a set of findings. All findings are supported by statements of multiple participants.

Quantitative Study (Survey) **Protocol.** In the second step of our study, we designed a survey with 32 questions, most of which map to our initial findings. The questions were primarily formulated as statements, asking participants to state their agreement on a five point Likert scale (examples of these questions can be seen in Figure 2.3). To target developers with experience in cloud technologies, we gathered profiles of GitHub[3] users that "follow" a repository of a number of popular cloud platforms, including Amazon Web Services, Heroku, Google Cloud Platform, CloudFoundry, and OpenStack. We then discarded all users without a public email address in their profile, and contacted the remaining users with a description and link to our online survey. To motivate developers to participate in the survey, we held a raffle for all participants to win one of two 50 USD Amazon gift vouchers. The survey was in English, and took an average of 12.2 minutes to complete.

Participants. We emailed the survey invitation to 2000 GitHub users and gathered a total of 294 responses (response rate of 14.7%). Of all 294 participants, 192 were employed in smaller companies (between 1 and 100 employees) and 102 were employed in large companies. 71% stated their job role as software developer, 22% as team lead or product owner, 4% as operations engineer, and the remaining 3% listed software architect, researcher or chief technology officer

[3]http://www.github.com

(CTO). The average professional development experience per participant was 9 years (± 5).

Analysis. We analyzed the response distributions and present the results along with the findings from the interview study phase. Furthermore, we examined the responses of three free-text questions on overall differences in development in cloud versus non-cloud environments, restrictions in the cloud, and tooling. Based on these results, we were able to enhance our understanding of some of our findings.

Qualitative Deep-Dive Interviews (Interview$_2$) **Protocol.** Through the examination of open-ended survey questions, we identified 4 topics of high interest: *(1) fault localization, (2) monitoring and performance troubleshooting, (3) cost of operation,* and *(4) design for scalability.* In order to get more profound insights into these topics, we defined an overall set of 27 questions for these 4 topics and conducted another round of semi-structured interviews. We followed the same protocol as in the first round of qualitative interviews. Interviews lasted between 30 and 45 minutes and were audio-recorded.

Participants. Interviewees were recruited through our personal network. Overall, we recruited 9 participants (D1 to D9), 8 male and 1 female software developer with an average professional development experience of 8 years (± 6) from 6 different countries and two continents. All participants were from different companies. 3 participants deployed on PaaS, 5 on IaaS and one on IaaS but also on PaaS.

Analysis. After the interviews and based on the topics and categories identified in the previous two steps, we used open coding to categorize interview statements and gather more profound insights into the difference between cloud and non-cloud development.

2.5 Findings

In the following, we present the findings of our study. We first give a high-level overview of our findings in Section 2.5.1, and then provide more detailed results relating to **RQ1** (Section 2.5.2) and **RQ2** (Section 2.5.3).

2.5.1 Overview

For our study, we were primarily interested in examining what makes software development for the cloud "unique", i.e., what differs in terms of processes, tools, and implementation choices, to other development projects. However, early on in our interviews, it became clear that "the" cloud does not exist for practitioners. Hence, we asked our survey respondents to list the main characteristics that define cloud computing for them and gathered answers from 160 developers (multiple answers were allowed). We categorized and quantified the most common themes, and present them in Figure 2.2. Note that the last three entries in the chart—automation, ease of infrastructure maintenance, and elasticity—are all strongly related, as are the first entries in the chart—focus on product, faster time-to-market.

The majority of developers thinks about the cloud mostly as a deployment and hosting technology, following either the IaaS or PaaS model. For these developers, the ability to easily scale applications and the ease of infrastructure maintenance is what makes the cloud unique. While productivity and faster time-to-market has been named as a distinguishing feature in our interviews, these were only relevant to about every tenth survey respondent. Finally, it is interesting to note that reduced Total Costs of Ownership (TCO) are also only a distinguishing factor for a minority of cloud developers.

Our interviews further revealed that there is a large mindset gap between developers who think of the cloud as either IaaS or PaaS, and those who think of it mostly in terms of SaaS. For developers where cloud is seen more as a delivery model (i.e., SaaS), how the application is actually hosted tends to be opaque in cloud and non-cloud environments alike and not much changed when adopting cloud computing.

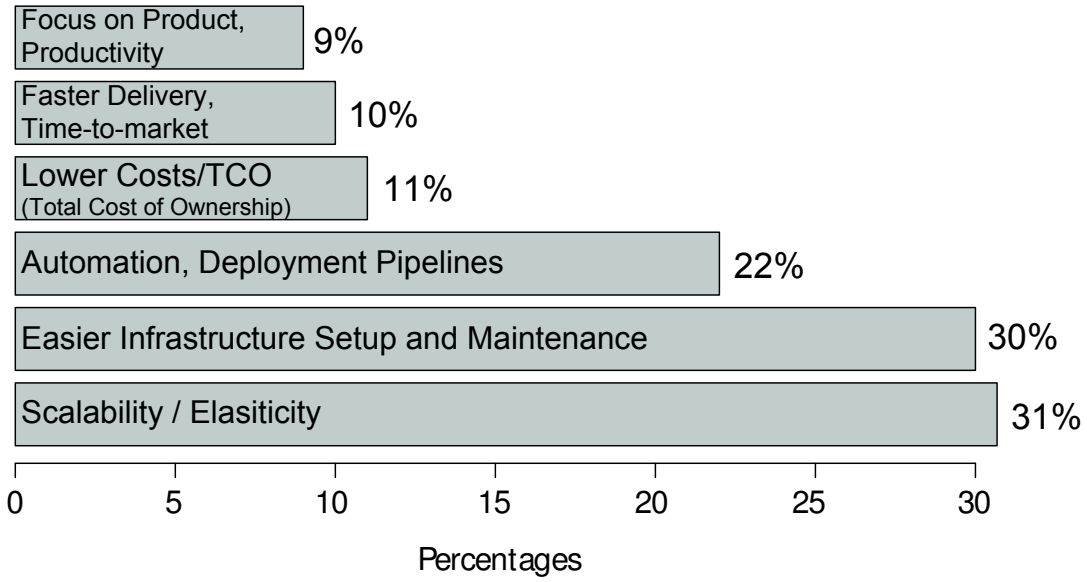


Figure 2.2: Main Differences in Cloud Development

This was, however, different for interview participants working on IaaS and PaaS services. These participants consistently commented on a range of differences. The analysis of our interview transcripts and the survey data shows that the changes when adopting cloud computing for IaaS and PaaS developers fall into the following broad categories: changes to how applications are provisioned and deployed (**Deployment & Automation**), changes in how applications are actually built (**Design & Implementation**), changes in how problems can be debugged (**Troubleshooting & Maintenance**), and cultural changes (**DevOps Communication**). Other areas that are part of software engineering, such as requirements engineering or security, are not addressed in this study as they did not emerge from our study data.

We were also interested in investigating the role of data and operational metrics, as it plays an important role in software development [Buse and Zimmermann,]. Indeed, the analysis of our study showed that it plays a major role in cloud development.

Concretely, the usage of cloud-based tooling has increased (**Cloud Tooling**), more and more types of data are available for teams (**Monitoring & Produc-**

Figure 2.3: Results from our quantitative survey

tion Data), but that developers currently struggle to fully utilize this additional data (**Usage of Runtime Data**). In the following, we will discuss our detailed findings based on these broad groups of changes.

In Figure 2.3, we summarize the quantitative results for the Likert-type scale questions from our survey that relate to our core research questions. We grouped the results in Figure 2.3 according to our subtopics. The insights from *Deployment & Automation* and *Troubleshooting & Maintenance* resulted primarily from our interviews, rather than the survey.

In general, we saw a high level of agreement with our interview findings. However, a few individual questions showed some disagreement as well, requiring more detailed study. These aspects are discussed in more detail in the remainder of the paper. In the following we present the main themes of our study. Where applicable, we provide quantitative results from our survey in the presentation of our findings.

2.5.2 Application Development and Operation

In this section, we report on how cloud computing has affected application development and operations, as well as on the main drivers for these changes, API-driven infrastructure-at-scale and cloud instance volatility.

Deployment & Automation

Our interviews have shown that elasticity, ease of infrastructure maintenance, and automation can be broken down into two fundamental aspects that drive most changes of software development in the cloud: (1) *API-driven infrastructure at scale* and (2) *volatility of cloud instances*. Both these aspects have ripple effects on almost every aspect of cloud development.

API-driven Infrastructure-at-Scale. Infrastructure-at-scale refers to the ability to quickly spawn up (and discard) many compute instances using an API. This ability requires more automation on many levels, including infrastructure, environment and test. In *IaaS clouds*, automation happens by defining your

infrastructure as a set of software artifacts (see IaC in Section 2.2). This allows for automatic provisioning of newly created instances for different purposes (e.g., scaling up, setting up a test environment). In PaaS clouds, applications are required to be packaged in a way to be easily reproducible (e.g., containers, buildpacks) to manage this automation behind the scenes. All interviewees deploying on IaaS agreed that the use of IaC tools (e.g., Chef, Puppet) or other means of automation (e.g, shell scripts) for all automated provisioning of their infrastructure has become essential in the cloud.

Volatility of Cloud Instances. Cloud instances can be started up and shut down for various reasons. This volatility happens either through (1) the cloud provider shutting down instances, (2) the load balancer spawning or shutting down instances, and (3) the application itself shutting down a dynamically allocated instance that finished its work. One implication of instance volatility is that all infrastructure definition and server configuration has to be implemented in code. If provisioning and configuration is not automated, it is bound to be lost when instances are discarded. This means that every change in infrastructure results in a new deployment of the system. Four of the interviewees that deploy in IaaS referred to this practice as *immutable infrastructure*:

> "*We have now moved to strict Immutable Infrastructure in our deployment. We don't even put SSH keys into instances anymore, making changes to existing infrastructure impossible*" -P12$_{IaaS}$

In PaaS, the infrastructure is managed by the cloud provider. Hence, the infrastructure is immutable for developers by default.

Infrastructure Transparency. All interviewees deploying on IaaS mentioned the use of either IaC tools or shell scripts for all automated provisioning of their infrastructure. They argue that this brings them transparency regarding their infrastructure:

> "*What happens in our infrastructure is a lot more obvious. Everything we do on that level [infrastructure] is over code (...) So, I don't need ask my colleague what he did to get that process running - I just look at the code and maybe the commit history*" -P9$_{IaaS}$

Virtual Containers for Automation. Furthermore, three interviewees describe a push in virtualization from virtual machines towards virtual containers (e.g., LXC[4], Docker[5]) for their automation:

> "*Virtual machines are too slow in a large scale (...) Speed matters, also in integration testing. When I can make a build take 3 minutes instead of 20, that's a huge win*" -P13$_{IaaS}$

Traditional virtual machines impose a large performance overhead due to the additional virtualization layer. Containers allow for much faster start-up times and are, therefore, also increasingly used as a base for PaaS [Tang et al., 2014].

Infrastructure provisioning and application deployment in the cloud are largely automated. Servers are not seen as durable entities. Hence, any changes to infrastructure need to be defined in code. This also leads to more transparency concerning infrastructure changes.

Design & Implementation

In this section, we report on how restrictions in the cloud influence how developers design and implement their applications.

Design Restrictions. As part of our survey, we asked whether the survey respondents face limitations in application architecture and design specific to the cloud. Results to this question were inconclusive, with 51% of respondents

[4] `https://linuxcontainers.org/`
[5] `https://www.docker.com/`

disagreeing and 36% agreeing (see Q5 in Figure 2.3). However, 119 respondents have still (in a free-form field in the survey) stated multiple restrictions, which we categorized and quantified in Figure 2.4.

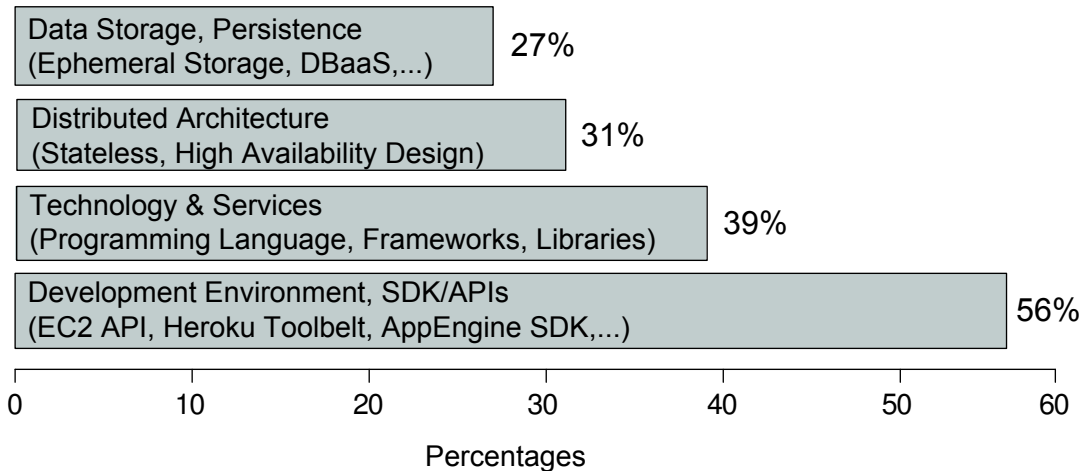


Figure 2.4: Most significant restrictions developers have encountered on cloud platforms

It is not surprising that there are technical restrictions regarding the supported SDKs, libraries, and frameworks. However, to our interview participants, these restrictions were not all negative. Some developers feel that technological restrictions allow them to focus more on delivering business value, instead of tinkering with low-level technology choices:

> "*But I don't wanna make those decisions [on technology] anyway. (...) The cool thing - from a product design point of view - is, that you know what works on AppEngine and what doesn't*" -$P11_{PaaS}$

More interesting is that close to a third of all respondents also consider the cloud to restrict them in the way they actually architect and design their applications. These restrictions in design and architecture are primarily caused by

API-driven infrastructure-at-scale and volatility of cloud instances, as discussed in Section 2.5.2.

Design for Failure. Volatility of cloud instances naturally forces developers to build highly fault-tolerant applications, as IaaS providers reserve the right to shut down any resources at any time, on short or without any prior notice:

> "*One interesting thing that is very cloud specific and influenced our architecture is, that the cloud provider tells you, we can kill your machine any time we want.*" -$P9_{IaaS}$

Well-known cloud users have already adopted this mindset. Netflix, for instance, has stated that they use an application called Chaos Monkey[6] to randomly terminate cloud instances in production, to force application design that can tolerate such failures when they happen unintentionally.

Design for Scalability. Scalability is the most named difference in cloud development for survey participants (see Figure 2.2). In our deep-dive interviews we asked participants to explain how scalability considerations influenced them during design and implementation.

All interviewees stated that they always have scalability in mind, even when designing very simple cloud applications:

> "*Even if the customer needs only 1 server, we have an ELB (Elastic Load Balancer) anyway, because we expect everything to grow*" -$D3_{IaaS}$

The Twelve-Factor app[7] design has become the de-facto standard when it comes to best practices when building cloud applications. A few of the interview and survey participants referred particularly to this manifesto, while others often referenced similar practices for implementing scalability under a different name.

[6] `http://techblog.netflix.com/2012/07/chaos-monkey-released-into-wild.html`

[7] `http://12factor.net/`

An alternative approach to implement fault-tolerant and scalable cloud applications that was mentioned are microservices [Newman, 2015]:

> "*We have divided our application into many services. For one specific service we kill off and start new instances all the time, also to have proper redundancy.*"
> -$P9_{IaaS}$

However, in our interviews, only five participants mentioned either currently using or having plans to move to a more service-oriented system design in the near future.

> The cloud imposes some restrictions on how applications can be built, in technological and software architectural terms. Specifically, applications need to be designed for scalability and fault tolerance. These restrictions are also seen as positive as they enforce the best practices and foster business value.

Troubleshooting & Maintenance

Activities that happen after the application has been deployed (i.e., troubleshooting and maintenance) have probably seen the biggest change in cloud development. Servers are not seen as durable entities anymore. Therefore, infrastructure maintenance has become an activity that now has to deal with adapting infrastructure code files rather than tweaking on live server instances.

Fault Localization. Fault localization, the activity of discovering the exact locations of program faults, cannot be done through ad-hoc inspection on, for instance, log files, memory dumps, or system metrics on live production servers anymore.

This means that every information that is of interest in the maintenance phase of the development life cycle must be specified before deployment and collected in a central repository, otherwise information runs the risk of being lost due to cloud instance volatility. These techniques are not necessarily bound to the cloud. However, in the cloud these best practices are seen as mandatory:

> "*In the cloud you are forced to use best practices, you are forced to use automation. You are forced on not relying to having root access to jump onto a machine and search logs by hand. You are forced to use better practices like aggregation.*" -$D6_{PaaS}$

Reproducing Issues. In terms of reproducibility of issues in a local environment for fault localization, our interviewees were somewhat divided. On the one hand, this task has become more difficult, as cloud applications are inherently distributed and reproducing a distributed environment locally is generally a difficult task. On the other hand, deployment automation makes it easier and faster to spin up a staging or testing environment in the cloud to reproduce issues. However, our interview partners also mentioned the additional cost involved of spinning up environments as a reason for first attempting to reproduce issues locally. Some interviewees also stated that proper end-to-end monitoring [Cito et al., 2014a, Cito et al., 2015a] and request tracing needs to be in place to be able to reproduce issues correctly:

> "*If you missed some logs and the instance is already gone, have fun reproducing your environment*" -$D8_{PaaS}$

Especially with public cloud providers, hardware characteristics also need to be tracked, as you never know what specific hardware configuration will be served [Leitner and Cito, 2014].

Troubleshooting has changed, as problematic cloud instances are often not accessible or already discarded, rendering hot fixes or searching for logs in production impossible. Instead, relevant logs must be defined beforehand and collected in a central repository. These best practices are not exclusive to the cloud, but instance volatility makes their usage in cloud computing mandatory.

DevOps Communication

As already discussed in Section 2.2, it is often argued that cloud computing goes hand in hand with a DevOps style of communication, which leads to higher collaboration between software developers and operations engineers. In our interviews, this notion was not undisputed. While 12 of the interviewees have agreed with the DevOps vision, the remaining participants (mostly from enterprise companies) have argued that there are still dedicated development and operations teams that more or less work as silos. Even companies that self-identified as following a DevOps approach still seem to generally have a separation between engineers that solely implement functional features, and engineers that mostly develop infrastructure code:

> "*We have our server/DevOps guy. (. . .) He handles the whole monitoring and tools thing*" -$P9_{IaaS}$

In our survey, there was a general agreement with these observations (see Figure 2.5). The survey responses show that, especially in large companies, communication and interaction has increased between application developers and operations engineers (72%). Contrary to our interview study, the difference between the responses of developers working in smaller or larger companies in terms of collaboration between development and operations is not large. For smaller companies, 70-74% tend to agree that their operations and development are now handled by the same staff, versus 57-60% in larger companies. This data suggests that, even in larger companies, the gap between development and operations activities converges.

Close to 60% of developers across companies of all sizes build applications following the DevOps notion of converging development and operations teams. However, even in a DevOps team, there are still dedicated engineers that are responsible for maintaining the infrastructure code.

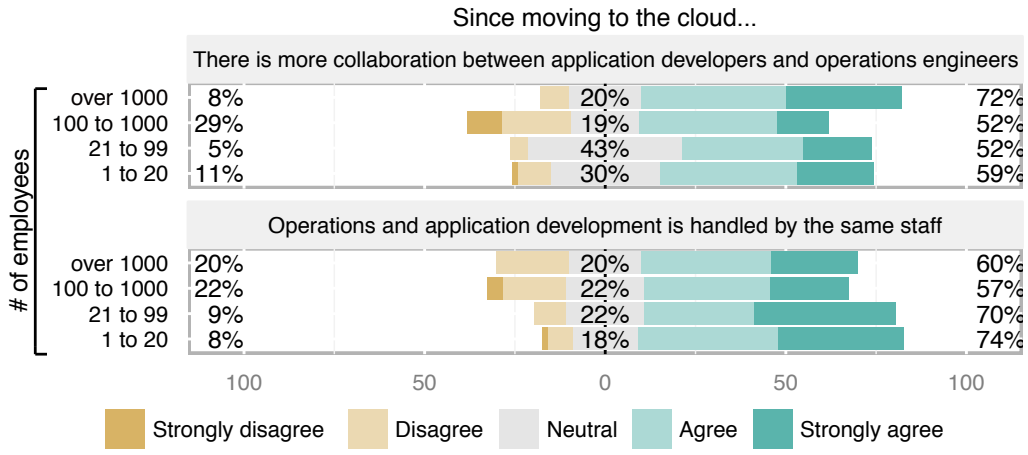


Figure 2.5: Results regarding Team Communication grouped by company size

2.5.3 Changes in Tools and Metrics Usage

In our research, we were particularly interested in how the usage of tools and production data has changed in cloud development projects. We report on (1) how tooling has evolved in the cloud, (2) what kind of metrics are available now, and (3) how these metrics are utilized.

Cloud Tooling

Tooling for the Cloud. In the survey, we asked what tools developers specifically use in development for the cloud, which they have not used before. 124 people responded to the question with multiple tools, which we categorized and quantified in Figure 2.6. We also asked whether cloud-based tool usage has generally increased. 67% of respondents agreed with this statement, while 15% disagreed (see Q6 in Figure 2.3).

Performance and Log management top the list of tools survey respondents use specifically for cloud development. This can again be attributed to the notions of API-driven infrastructure-at-scale and cloud instance volatility, which we have already discussed extensively in Section 2.5.2.

Tooling in the Cloud. We also observe an increase in the usage of tooling that is itself cloud-based. However, we cannot differentiate whether this rise

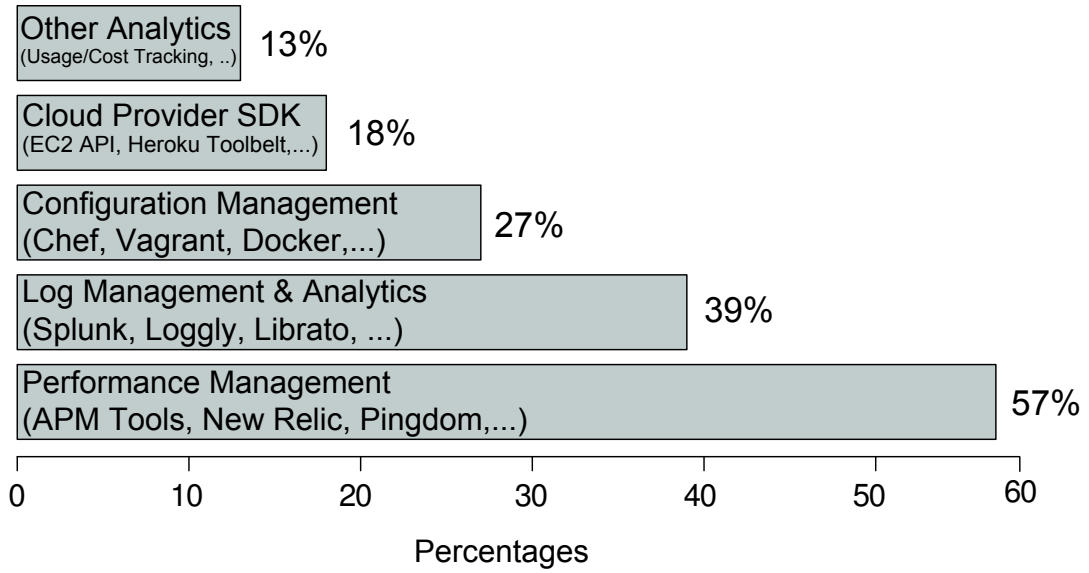


Figure 2.6: Tools used specifically for cloud development

has to do with development on cloud platforms, or with the fact that more and more tooling is moving to a SaaS model in general. Interviewees agreed that, when building cloud applications, many of the tools previously installed in the local infrastructure or on the developer machine are now also in the cloud. This includes tools for monitoring, analytics, and configuration management. Four of the interviewees even used an entirely Web-based IDE for most of their development tasks. All others moved at least part of their supporting tooling into the cloud. The interviewees developing entirely in the cloud (i.e., through a cloud-based IDE) expressed the overall relief of not having to maintain or configure a local development environment:

> "*I don't need to take care of backups, or updates of the IDE. I can use the same setup everywhere. (...) Also there's no hassle about that missing plugin and that wrong configuration.*" -P1$_{PaaS}$

However, cloud-based IDEs were not mentioned by survey respondents at all as part of their tooling. Hence, we conclude that cloud-based IDEs have, unlike indicated by our interview study, not yet found mainstream adoption.

> Cloud-based tool usage has generally increased. Performance, log management and analytics tools are most used specifically for cloud development. Many of these tools are themselves cloud-based. However, despite some advantages, cloud-based IDEs have not yet found mainstream usage.

Monitoring & Production Data

Metric Awareness. All interview participants have mentioned using solutions for log aggregation and central operational metric collection for cloud development. The survey supports these claims with performance metric and log management tools being deemed the most important. What we have also observed is that the enforcement of these practices has led to more awareness for metrics:

> "*Software Developers have not - until recently - seen metrics as very important. They saw that stuff as an operations thing*" -$D5_{PaaS}$

In our survey, 72% of cloud developers agree that they look at more metrics on production systems than before (see Q8 in Figure 2.3).

Metric Availability. In general, interviewees expressed that they now have a much richer set of monitoring metrics on production systems available:

> "*Especially now in the cloud (with Heroku or Amazon) they provide so much data. Putting this data in a graphing tool and looking at it once in a while has become increasingly easier.*" -$P12_{IaaS}$

From the survey, we see that 62% of respondents agree that they actually have more operational metrics available than before (see Q7 in Figure 2.3). When asked

whether access to production data has become easier, survey participants were divided: 21% disagreed, 38% were neutral and 41% agreed with the statement (see Q9 in Figure 2.3).

We investigated this claim further and found that metric utilization has increased both in quantity and dimension. In addition to technical system-level metrics (e.g., CPU utilization, cache hits), more teams now look also at business metrics (e.g., customer retention, number of logins) to guide their decisions. However, system performance metrics are deemed to be the most interesting by our survey participants. 84% state that they look at performance metrics on a regular basis (see Q10 in Figure 2.3).

> Cloud developers look at more production metrics. In addition to system-level metrics, business metrics are becoming more relevant. However, application performance is most often still measured via system-level metrics.

Usage of Runtime Data

An increase of tooling to acquire data, more metric awareness and availability spiked our interest in how developers use metrics in their work. Through analysis of our initial interviews and the open questions in our survey, we identified that *performance* and *cost* were the metrics of high interest to cloud developers. In the following we describe how our interview participants utilize this data in their regular development work.

Performance Troubleshooting. In our deep-dive interviews, we investigated how our study participants approach a particular performance problem by utilizing data from production systems to solve the problem.

A common theme is that developers follow a more reactive approach to metrics, i.e., they act on alerts or reports on the issue tracker provided by someone who is concerned with running the application in production (i.e., DevOps engineers, operations engineers, or system administrators). Metrics are typically provided in dashboards, accessible to everyone in the team, but mostly used by operations. However, when actively debugging a known performance issue,

all interview participants would first go *"by intuition"* and reproduce the issue in their own development environment, before inspecting how metrics have evolved in production in the provided dashboards. Only if the local inspection does not yield any results, they would dig deeper into production data. Interviewees stated that they choose to ignore the data at first, because it is rather cumbersome to navigate performance dashboards, while at the same time navigating through code:

> "*I try to reproduce and solve the issue locally. Looking for the particular issue in the dashboard and jumping back and forth between the code is rather tedious*" -$D2_{IaaS}$

Costs of Deployment. To most of our interviewees, the costs of deployment in the cloud were generally deemed as important. However, developers seem to only in the abstract be aware that their design and implementation decisions have an influence on deployment costs. When confronted on how the costs of deployment (especially of specific code changes) are used in their daily work, it became clear that costs are not a tangible factor for developers. Some interviewees argued that having this information more wide-spread and accessible would be interesting to them, but does not fall into their responsibilities:

> "*I have no idea about the costs. I can just read it in the logs sometimes that in production we spawned 20-30 servers. It would be interesting to know, but it's not really important for application development*" -$P5_{PaaS}$

Not developers, but software architects or the CTO are concerned with the overall costs of operation. However, even for these roles, costs were considered in a post-design phase and currently do not influence their design decisions directly.

Currently, developers struggle to use the abundance of available runtime metrics. Rather, they often solve performance problems "by intuition" in a local environment. More detailed inspection of metrics is only used when this approach does not lead to a solution. Cloud costs are deemed as important, but are not tangible to developers.

2.6 Discussion

We presented the results of the first systematic study on how professional software developers build applications on top of cloud infrastructures or platforms by addressing two research questions. We briefly revisit these research questions before discussing the ramifications of our results.

***RQ 1**: How does the development and operation of applications change in a cloud environment?*
In the cloud, servers are *volatile*. They are regularly terminated and re-created, often without direct influence of the cloud developer. Our study has shown that the concept of API-driven infrastructure-at-scale and the cloud instance volatility have ripple effects throughout the entire application development and operations process. They restrict the design of cloud applications and force developers to heavily rely on infrastructure automation, log management, and metrics centralization. While these concepts are also useful in non-cloud environments, they are mandatory for successful application development and operation in the cloud.

***RQ 2**: What kind of tools and data do developers utilize for building cloud software?*
Based on our research, more data, and more types of data, are utilized in the cloud, for instance business metrics (e.g., conversion rates) in addition to system-level data (e.g., CPU utilization). However, developers struggle to directly interpret and make use of this additional data, as current metrics are often not actionable for them. Similarly, cloud developers are in the abstract aware that

their design and implementation decisions have monetary consequences, but in their daily work, they do not currently think much about the costs of operating their application in the cloud.

We now present, in a condensed form, the implications of our study results for practitioners, as well as the main challenges that cloud developers face. These form open problems that academic research and vendors of cloud-related tooling need to address to improve the experience of developers.

2.6.1 Implications for Practitioners

Our study has shown that there are a number of best-practices for building useful applications on top of IaaS and PaaS systems. These practices are not necessarily bound only to cloud development, but the nature of cloud infrastructures make these practices central for successfully deploying in a cloud. In the following we present a set of guidelines for software development in the cloud resulting from the findings of our study.

Cloud instances are *volatile*. Never assume that any instance will exist forever. In both, IaaS and PaaS cloud systems, backend instances come and go. Logs, configuration changes, or hot fixes stored only on a cloud instance are bound to be lost. As we have seen in Section 2.5.2, cloud developers should treat cloud instances as *immutable* black boxes, which they in general cannot fix, and in some cases cannot even log into. This requires a change in mindset for engineers used to having full control over their infrastructure.

***Anticipate runtime problems* and define relevant logs and operational metrics for fault localization before deployment. Use log management tools to centrally collect this data.** Related to the volatility of cloud instances, developers need to start thinking about how to localize and debug runtime faults already during development. This includes defining useful debugging statements, logs and operational metrics prior to deployment, as well as setting up and using tools to centrally *collect, persist, and analyse* these metrics outside of volatile instances. As discussed in Section 2.5.2, cloud instances are black boxes, drilling into unanticipated problems after deployment is often

impossible. Hence, cloud developers should be aggressive in what they log and what operational metrics they trace. It is easier to filter out data that turns out to be unnecessary than to debug problems for which the relevant logs and metrics have not been collected.

Scalability and fault tolerance need to be *first-class citizens* in application design. While most distributed and Web-based applications have historically striven to be scalable and fault tolerant, at least to some degree, these concepts have become even more central in cloud
projects. API-driven infrastructure-at-scale means that essentially any application or component can scale up dynamically (e.g., to react to increased load). Instance volatility means that runtime faults are bound to happen. As such, scalability and fault tolerance need to be first-class citizens when designing applications (see Section 2.5.2). *Best-practices for cloud application design* (e.g., the 12-Factor App) take this into account and should not be compromised by cloud developers. PaaS systems often enforce such application design through restrictions (e.g., in terms of statelessness). Developers should not aim to circumvent, but rather embrace those restrictions.

For IaaS, *automate* server provisioning and configuration, for instance using IaC tools. API-driven infrastructure-at-scale requires that the process of instance provisioning and configuration is fully automated. Besides being able to scale up quickly, this also has the additional advantage that knowledge about how servers are configured is explicitly documented in scripts or IaC code, and versioned in the project's version control system. This allows developers to *revert*, for instance, erroneous changes in the configuration of a cloud instance just like they would revert a broken application code change. It also makes infrastructure configuration and evolution explicit for other developers and DevOps engineers leading to more *transparency*, as discussed in Section 2.5.2. While some cloud developers currently use scripting (e.g., `bash`) for this purpose, the usage of dedicated IaC tools has additional advantages. Most importantly, IaC allows for reuse of existing open source provisioning and configuration code, and supports unit testing well.

Embrace the *tools and data* the cloud provides you. As elaborated in Section 2.5.3, we have seen that cloud developers have access to more tools and data. However, we have also seen that many developers still rather go *"by intuition"* when debugging problems rather than analyzing provided operational metrics. We argue that, besides better tooling (see Section 2.6.2), a change in mindset is required for cloud developers to fully embrace the additional options for debugging and maintaining applications that the cloud environment provides them.

2.6.2 Challenges for Cloud Development

In addition to the best-practices and guidelines outlined in Section 6.5, which cloud developers can already implement today, we have also seen that there are a number of areas in which academic research and tool vendors should provide better techniques, approaches and tools.

Academic Research on Infrastructure Evolution. In IaaS clouds, software developers make use of scripting or IaC tools to formally specify and track infrastructure configuration. This means that infrastructure evolution is now tracked in version control systems the same way the evolution of regular code artifacts is. We argue that this provides *new opportunities for academic research* on software repository mining to investigate how infrastructure code evolves over the lifetime of a project, compared to the overall application code. This will allow us to, for instance, discover anti-patterns in infrastructure provisioning code.

Improved Log and Metrics Management. Cloud developers need to anticipate problems prior to deployment and define relevant logs and operational metrics. While this is largely already possible today, there is little support to guide developers *what* and *how* they should be logging exactly. Some study participants have reported that this results in rather excruciating trial-and-error. We argue that correct tracing and metric definition has to be introduced as part of the development workflow. Methods and tooling should be improved to support the process of defining and evolving useful logs and metrics for cloud applications.

Local Reproducibility through Containers. Cloud applications are distributed by default. Our study participants reported that trying to *reproduce faults locally* can be a tedious and time-consuming task. It involves knowing the exact state of when the fault occurred in production (i.e., infrastructure and data state), as well as the capability to replicate the environment and its state locally. We envision methods and tools that have the ability to recreate a distributed environment in a local development environment from a snapshot of when the fault occurred, utilizing containerization technology (e.g., using Docker).

Tools for Developer Targeted Analytics. In the cloud, more metrics are available, both in quantity and dimension. However, in our study we have observed that developers, before utilizing existing metrics, much rather rely on their experience and intuition to solve problems. Besides a required change in mindset, as discussed in Section 6.5, we attribute this to the fact that most monitoring tools are built to be used by operations teams, rather than software developers. Currently, the existing way of delivering metrics is difficult to leverage for developers as it is *not actionable in the development process*. Existing tools need to expose better APIs for data extraction and integration. Future research needs to address how the abundance of data in the cloud can become more actionable for developers and integrate it into their daily workflows [Cito et al., 2015a]. Possible solutions could include integration of data in code views through IDE plugins or within issue tracking systems.

2.7 Threats to Validity

While we have designed our research as a mixed-mode study to reduce threats to validity as far as possible, there are still a number of limitations inherent in our research design. Primarily, the question arises to what extent the 25 cloud developers we interviewed are representative (*external validity*). However, to mitigate this threat, we have made sure to recruit interview participants that are approximately evenly distributed between smaller companies and larger enterprises, as well as between IaaS and PaaS clouds. There still are a more interviewees that deploy on PaaS than on IaaS. We think to have mitigated

this concern by having more IaaS participants in the survey, balancing the overall results. Further, our interview participants cover a broad range of experience levels, and work with various kinds of cloud systems on various kinds of applications. A similar threat also exists for the external validity of our survey. We recruited our participants almost exclusively via GitHub, meaning that we are likely to have attracted mostly software engineers who are actively interested in open source development, and who are also following the progress of at least one big open source cloud product. Further, responses were necessarily voluntary, hence we are likely to have attracted a crowd with higher-than-average interest in the topic of cloud computing.

In terms of *internal validity*, it is possible that we have biased our interview partners through the pre-selection of questions and topics, and that we missed entirely unanticipated differences and implications of software development in the cloud. Also, our themes focused on how software development in the cloud is different than in non-cloud environments. Thus, interview participants may have been inclined to overstate differences, leaving out similarities. However, given that no major undiscovered differences and implications were mentioned during the survey either, we judge this threat to be low.

2.8 Conclusions

We report on the first systematic study on how professional software developers build applications and utilize specialized tools and data on top of cloud infrastructures and platforms. The insights provided by our study help to better understand how cloud computing has made an impact throughout the software development life cycle. Our findings suggest, among others things, two major developments: (1) developers need better means to anticipate runtime problems in the cloud and rigorously define and collect metrics for better fault localization and (2) the cloud offers an abundance of operational data, however, developers still often rely on their experience and intuition rather than utilizing metrics to solve problems. Methods and tools for developers will, therefore, need to adapt to these required changes in the cloud. From our findings, we extracted a set of

guidelines for cloud development and identified challenges for researchers and tool vendors to support developers in these efforts by developing new approaches for managing metrics and making operational data more actionable.

3

Runtime Metric Meets Developer

***Jürgen Cito**, Philipp Leitner, Harald C. Gall,*
Aryan Dadashi, Anne Keller, Andreas Roth
Published in the Proceedings of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward! 2015)
Contribution: Framework design, prototype implementations, and paper writing

Abstract

A unifying theme of many ongoing trends in software engineering is a blurring of the boundaries between building and operating software products. In this paper, we explore what we consider to be the logical next step in this succession: integrating runtime monitoring data from production deployments of the software into the tools developers utilize in their daily workflows (i.e., IDEs) to enable

tighter feedback loops. We refer to this notion as *feedback-driven development* (FDD). This more abstract FDD concept can be instantiated in various ways, ranging from IDE plugins that implement feedback-driven refactoring and code optimization to plugins that predict performance and cost implications of code changes prior to even deploying the new version of the software. We demonstrate existing proof-of-concept realizations of these ideas and illustrate our vision of the future of FDD and cloud-based software development in general. Further, we discuss the major challenges that need to be solved before FDD can achieve mainstream adoption.

3.1 Introduction

With the widespread availability of broadband Internet, the software delivery process, and, as a consequence, industrial software engineering, has experienced a revolution. Instead of boxed software, users have become accustomed to software being delivered “as-a-Service" via the Web (SaaS [Turner et al., 2003]). By now, this trend spans various kinds of software, including enterprise applications (e.g., SAP SuccessFactors), office products (e.g., Windows Live), end-user applications (e.g., iCloud), or entire web-based operating systems (e.g., eyeOS). With SaaS, much faster release cycles have become a reality. We have gone from releases every few months or even years to weekly or daily releases. Many SaaS applications are even employing the notion of continuous delivery [Humble and Farley, 2010] (CD), where new features or bug fixes are rolled out immediately, without a defined release plan. In the most extreme cases, this can lead to multiple rollouts a day, as for instance claimed by Etsy, a platform for buying and selling hand-made crafts[1]. On the one hand, these circumstances have imposed new challenges to software development, such as the necessity not to postpone quality checks to a dedicated quality assurance phase, as well as necessitating a high degree of automation of the delivery process as well as cultural changes [Hüttermann, 2012, Chen, 2015]. On the other hand, SaaS and CD have also opened up tremendous new opportunities for software developers, such as to gradually rollout new features

[1] http://www.infoq.com/news/2014/03/etsy-deploy-50-times-a-day

and evaluate new ideas quickly using controlled experiments in the production environment [Kohavi et al., 2007].

Feedback-Driven Development

In this paper, we focus on one particular new opportunity in SaaS application development: tightly integrating the collection and analysis of runtime monitoring data (or feedback) from production SaaS deployments into the tools that developers use to actually work on new versions of the application (i.e., into Integrated Development Environments, or IDEs). We refer to this notion as *feedback-driven development* (FDD). FDD includes, but goes way beyond, visualizing performance in the IDE. We consider FDD to be a logical next step in a long succession of advancements in software engineering that blur the traditional boundaries between building and operating software products (e.g., cloud computing [Buyya et al., 2009], DevOps [Hüttermann, 2012], or live programming [McDirmid, 2007]).

We argue that now is the right time for FDD. Firstly, driving software development through runtime feedback is highly *necessary*, given that the fast release cycles prevalent in SaaS and CD do not allow for long requirements engineering and quality assurance phases. Secondly, the necessary feedback data is now *available*. Most SaaS applications are run using the cloud deployment model, where computing resources are centrally provided on-demand [Buyya et al., 2009]. This allows for central management and analysis of runtime data, often by using Application Performance Monitoring (APM) tools, such as New Relic[2]. However, currently, this runtime data coming from operations (*operations data*) is hardly integrated with the tooling and processes that software developers use in their daily work. Instead, operations data is usually available in external monitoring solutions, making it cumbersome for developers to look up required information. Further, these solutions are, in many cases, targeted at operations engineers, i.e., data is typically provided on system level only (e.g., CPU load of backend instances, throughput of the SaaS application as a whole) rather than

[2] `http://newrelic.com`

associated to the individual software artifacts that developers care about (e.g., lines of code, classes, or change sets). Hence, operations data is available in SaaS projects, but it is not easily *actionable* for software developers.

Contribution and Outline

In this paper, we discuss the basic idea behind the FDD paradigm based on two classes of FDD tools, namely *analytic* and *predictive* FDD. Analytic FDD tools bring runtime feedback directly into the IDE and associate performance data visually to the software artifacts that they relate to, hence making operations data actionable for software developers. For instance, this allows developers to refactor and optimize applications based on feedback on how the code actually behaves during usage.

Predictive FDD goes one step further, and warns developers about problems based on local code changes prior to even running the application in production. To this end, predictive FDD builds upon static code analysis [Cousot and Cousot, 2002, Ayewah et al., 2008], but augments it with knowledge about runtime behavior. This allows us to generate powerful warnings and predictions, which would not be possible based on static analysis alone. Even more sophisticated predictive FDD tools are able to use knowledge about the concrete system load that various service calls induce to predict the impact of code changes on the cloud hosting costs of the application, warning the developer prior to deployment about changes that will make hosting the application in the cloud substantially more costly.

In Section 3.2 we give a brief exposition of relevant background, which we follow up by an illustration of an example application that can benefit from FDD in Section 6.3. We introduce the general concepts in Section 3.4, and, in Section 3.5, substantiate the discussion using three concrete case studies of analytic as well as predictive FDD tools, which we have devised and implemented as part of the European research project CloudWave[3]. Further, we elaborate what major challenges remain that might impede the wide-spread adoption of feedback

[3]http://cloudwave-fp7.eu

usage in everyday development projects in Section 3.6. Finally, we discuss related research work in Section 3.7, and conclude the paper in Section 3.8.

3.2 Background

The overall vision of FDD is tightly coupled to a number of other recent advances in Web engineering, some of which we have already sketched in Section 3.1. We now provide a more detailed background, to allow the reader to understand what kind of applications we assume will be supported by our approach going forward. Specifically, three current trends (cloud computing, SaaS, and continuous delivery) form the basis of FDD.

3.2.1 SaaS and Cloud Computing

The concept "cloud computing" is famously lacking a crisp and agreed-upon definition. In this paper, we understand "cloud" applications to mean applications that are provided as a service over the Web (i.e., SaaS in the NIST model of cloud computing [Mell and Grance, 2011]), in contrast to applications that are licensed and installed on premise of a customer's site, or downloaded and run on the user's own desktop or mobile device. Figure 3.1 illustrates these different models.

SaaS has some interesting implications for the evolution and maintenance of applications. Most importantly, there is exactly one running instance of any one SaaS application, which is hosted by and under control of the service provider. This single instance serves all customers at the same time (multi-tenancy [Bezemer and Zaidman, 2010]). Implementing and rolling out new features in a SaaS environment is at the same time promising (as rollouts are entirely under the control of the service provider) and challenging (as every rollout potentially impacts each customer) [Bezemer et al., 2010]. Further, the SaaS model gives service providers ready access to a rich set of live performance and usage data, including, for instance, clickstream data, fault logs, accurate production performance metrics, or even production user data (e.g., uploaded

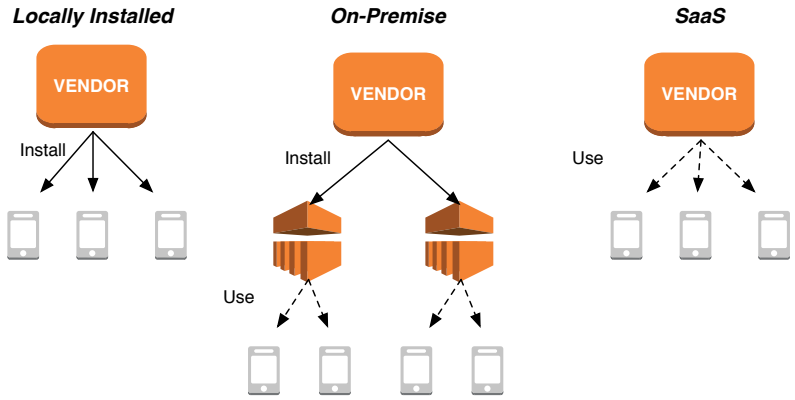


Figure 3.1: SaaS in contrast to on-premise or on-device software provisioning models. In SaaS, only one instance of the application exists and is accessed by clients directly. The software is never installed outside of the vendor's control.

videos, number and type of Skype contacts). In addition to supporting traditional operations tasks (e.g., application performance engineering), this abundance of data has also led to the ongoing "big data" hype [Bizer et al., 2012], which promises to generate deep market insight based on production data. However, these analyses are primarily on a strategic level (e.g., which product features to prioritize, which markets to address). Whether, and how, cloud feedback can also be used to support software developers in their daily workflow, i.e., while they are writing or optimizing code, is a much less discussed topic.

3.2.2 Continuous Delivery

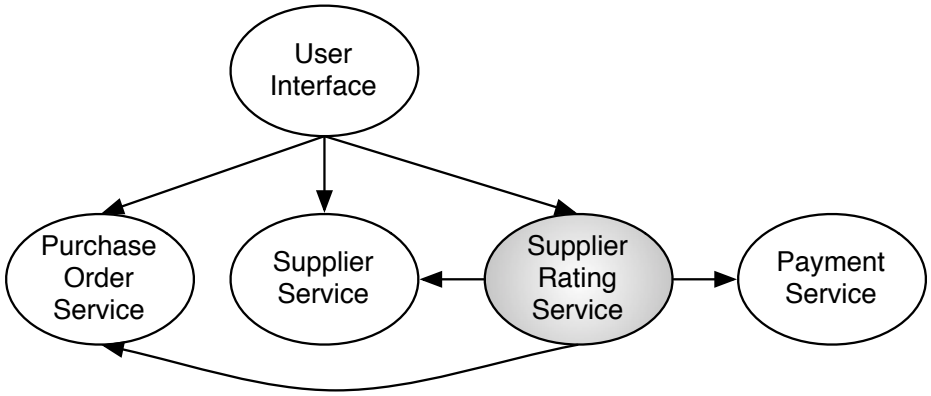
Another recent advance that is tightly coupled to the changed evolution of SaaS applications is continuous delivery (CD). CD has recently gained steam due to the success of companies such as Facebook, Google, or Etsy, all of which claim to employ CD to varying degrees for their core services. The most significant conceptual novelty behind CD is the abolishment of the traditional notion of releases. Instead, each change to the SaaS application may be shipped (i.e., pushed to production) independently of any release cycles "as soon as it is ready" [Humble and Farley, 2010].

In other release models (e.g., release trains [Khomh et al., 2012], as used by the Firefox project), new features are rolled out according to a defined release plan. If a new feature is not ready in time for a feature cut-off date, it is delayed to the next release, which may in the worst case take months. At companies like Etsy, features are rolled out as soon as they are deemed ready, independently of a defined release plan. Facebook claims to occupy a middle ground between release trains and strict CD, where most features are rolled out into production the same day they are ready, while a fraction of particularly sensitive changes (e.g., those related to privacy) are rolled out once a week [Feitelson et al., 2013]. In practice, these models result in frequent, but tiny, changes to the production environment, in the most extreme cases multiple times a day. This practice increases both, the business agility of the application provider, as well as the likelihood of releasing badly-performing changes to production [Spreitzer and Porath, 2012]. A consequence is that SaaS applications with such release models tend to be in a state of perpetual development [Feitelson et al., 2013] – there is never a stable, released and tagged version which is not supposed to be touched by developers.

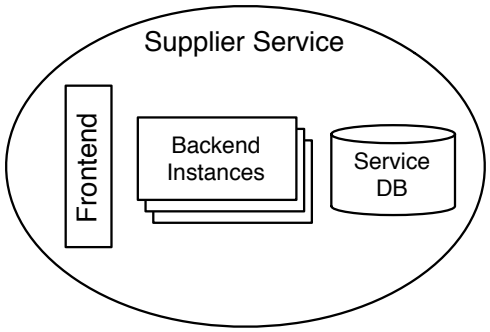
3.3 Illustrative Example

In the rest of this paper, we base our discussions on a (fictitious) enterprise application *MicroSRM*. *MicroSRM* is based on a microservice architecture [Newman, 2015, Schermann et al., 2015], of which a small excerpt is shown in Figure 3.2a. *MicroSRM* consists of a purchase order service (*PO*) and a supplier service (*Supplier*). The application allows companies to manage its purchases and suppliers, such that its business users can create, modify or delete purchase orders, or view individual purchase orders including all details, such as order items. Following the notion of microservices, each of those services consists of a frontend, which acts both as a load balancer and API for the clients of the service, a data storage, and a number of backend instances, which implement the main business logic of the service. The concrete number of backend instances can be adjusted dynamically,

based on load and following the idea of auto-scaling [Mao and Humphrey, 2011]. This is depicted in Figure 3.2b.



(a) Excerpt of the Microservices Architecture of *MicroSRM*



(b) Zooming into a Service *(SupplierService)*

Figure 3.2: Architecture overview of *MicroSRM* consisting of a network of Microservices.

Let us now assume that *MicroSRM* has been running successfully for a while, and the application is supposed to be extended with an additional service, a supplier rating service (*Rating*). The new service utilizes information accessed through the APIs of both, *PO* and *Supplier*. It calculates and rates how well suppliers have performed in the past, evaluating delivery performance, comparing prices, as well as user ratings persisted in the *Rating* service itself. After the

service has been deployed in production, it can be observed (through standard monitoring tools), that the new *Rating* service shows poor response time behavior, which had not been noticed through tests during development. The root cause for this performance problem has been that the complete data to recalculate the supplier rating based on orders from this supplier has been fetched at runtime from the *PO* service. With a growing number of orders this had slowed down the *Rating* service. Moreover, the algorithm used to calculate the rating was linear in the number of order items per supplier. To identify the root cause of the performance problem, the *Rating* service developers had to perform a manual analysis. They had to log the data volume accessed from the *PO* service and analyze the usage of this data inside the rating algorithm. Only then they could start to re-engineer the algorithm, replicate data from the *PO* service, or use a data aggregation API at the *PO* service.

At the time the performance problem was discovered and the analysis was conducted, it already had an impact on the end user. We argue that *feedback* on the application can be provided much earlier, already during the development of the *Rating* service. All operations data relevant for identifying the root cause of the performance problem (e.g. data volume inside the PO service) has already been available. It has just not been pushed to the appropriate level of abstraction, namely the development artifacts, such as the involved REST calls in the program code. Additionally, the available feedback has not been integrated into the daily workflow and tools of developers.

As we will detail in the following sections, FDD is about enabling the automation of this feedback process: aggregating operations data, binding it to development artifacts, and predicting and visualizing it in the IDE.

3.4 Feedback-Driven Development

In this section, we introduce our approach for *Feedback-driven Development* (FDD), a paradigm for systematically exploiting operations data to improve development of SaaS applications. We discuss that FDD is about tightly integrating the collection and analysis of feedback from production deployments into

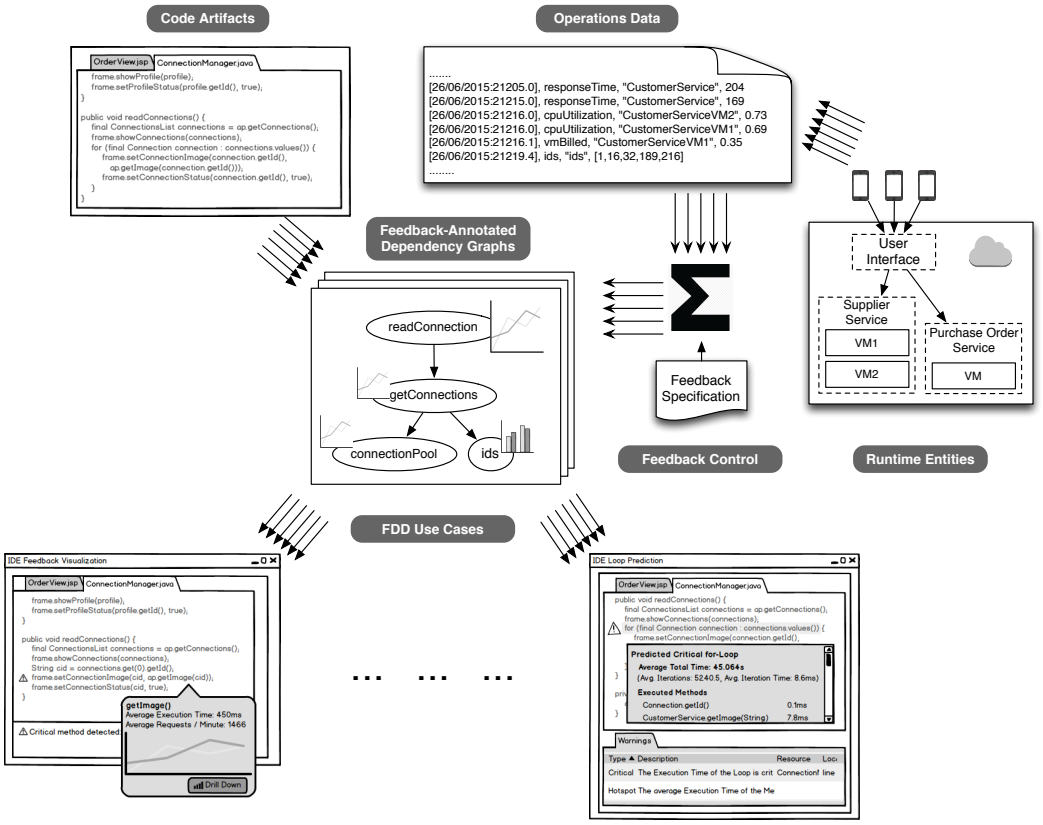


Figure 3.3: Conceptual overview of Feedback-Driven Development. Code artifacts are transformed into use case specific dependency graphs, which are enriched with feedback harvested in the production cloud environment. The annotated dependency graphs are then used to visualize feedback directly in the IDE, as well as predict the impact of changes to the program code.

the IDEs that developers use to actually work on new versions of the application. By nature, FDD is particularly suited to support more service-oriented and CD-based projects, but the underlying ideas are useful for the development of any kind of SaaS application.

FDD is not a concrete tool or process. Rather, FDD is an abstract idea, which can be realized in different ways, using different kinds of operations data and feedback, to support the developer in different ways. We refer to these different flavors of the same underlying idea as *FDD use cases*. In this section, we

discuss general FDD concepts, while concrete use cases and prototypical example implementations on top of different cloud and monitoring systems are the scope of Section 3.5.

3.4.1 Conceptual Overview

A high-level overview that illustrates the core ideas behind FDD is given in Figure 3.3. At its heart, FDD is about making operations data, which is routinely collected in the runtime environment, actionable for software developers. To this end, we transform *source code artifacts* (e.g., services, methods, method invocations, or data structures) into one or more graph representations modeling the dependencies between different artifacts (*dependency graphs*). Each node in these dependency graph represents a source code artifact $a_i \in \mathcal{A}$. The concrete semantics of these graphs, and what they contain, differ for different FDD use cases. For example, for feedback-based performance prediction, the dependency graph is a simplified abstract syntax tree (AST) of the application, containing only method definitions, method calls, and control structures (`ifs` and `loops`).

Operations data (e.g., service response times, server utilization, but also production data, such as the number of purchase orders as in our example) is collected from *runtime entities* (e.g., services or virtual machines running in the production environment). Every operations data point, d_i, is represented as a quadruple, $d_i = \langle t, \tau, e, v \rangle$, where:

- t is the time the operations data point has been measured,
- τ is the type of operations data (as discussed in Section 3.4.2),
- e refers to the runtime entity that produced the data point,
- v is the numerical or categorical value of the data point.

Operations data deriving from the measurement of a runtime entity usually come in the form of a series $\mathcal{D} = \{d_1, d_2, \ldots, d_n\}$. In our illustrative example, relevant runtime entities are, for instance, the services, virtual machines, and databases. Operations data can be delivered in various forms, for instance through execution

logs or events. *Feedback control* is the process of filtering, integrating, aggregating, and mapping this operations data to relevant nodes in the dependency graphs generated from code artifacts. Therefore, we define *feedback* as the mapping from source code artifacts to a set of operations data points, $\mathcal{F} : \mathcal{A} \mapsto \{\mathcal{D}\}$.

This process is steered by *feedback mapping*, which includes specifications (tailored to expected use cases) that contain the knowledge which operations data is mapped to which entries in the dependency graphs, and how. In our example, the services would need to be mapped to their invocation in the program code.

This *feedback-annotated dependency graphs* then form the basis of concrete FDD use cases or tools, which either *visualize* feedback in a way that is more directly actionable for the software developer (left-hand side example use case in Figure 3.3), or use the feedback to *predict* characteristics of the application (e.g., performance or costs) prior to deployment (right-hand side example).

3.4.2 Operations Data and Feedback

We now discuss the collection, aggregation, and mapping of operations data to feedback in more detail.

Types of Operations Data

In Figure 3.4, we provide a high-level taxonomy of types of operations data. Primarily, we consider *monitoring data*, i.e., the kind of operational application metadata that is typically collected by state-of-the-art APM tools, and *production data*, i.e., the data produced by the SaaS application itself, such as placed orders, customer information, and so on.

Monitoring data can be further split up into *execution performance data* (e.g., service response times, database query times), *load data* (e.g., incoming request rate, server utilization), *costs data* (e.g., hourly cloud virtual machine costs, data transfer costs per 10.000 page views), and *user behavior data* (e.g., clickstreams).

Feedback Control

All this operations data is, in principle, already available in today's cloud solutions, either via built-in cloud monitoring APIs (e.g., CloudWatch[4] in Amazon Web Services) or through external APM solutions. However, operations data by itself is typically not overly interesting to developers without proper analysis, aggregation, and integration. This is what we refer to as feedback control.

Feedback control is steered by feedback specifications, which are custom to any specific FDD use case. Further, software developers typically are able to further refine feedback specifications during visualization (e.g., via a slider in the IDE which controls the granularity of feedback that is visualized). Feedback control encompasses five steps, (1) data collection, (2) data filtering, (3) data aggregation, (4) data integration, and (5) feedback mapping, as depicted in Figure 3.5.

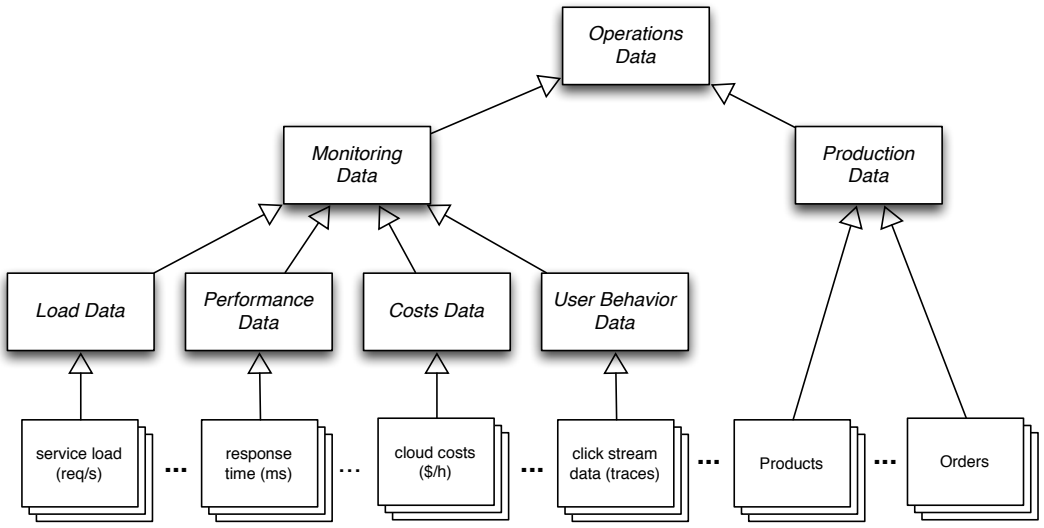


Figure 3.4: Overview of the types of operations data we consider in FDD and concrete examples. We distinguish between production data (i.e., the payload of the application, for instance, placed orders) and the monitoring information delivered by APM tools (e.g., response times or service load). A special type of APM data is usage data (e.g., click streams).

[4] `http://aws.amazon.com/cloudwatch/`

Data collection controls how operations data is monitored and collected on the target system. This may include instrumenting the application to produce certain operations data which would be unavailable otherwise (e.g., by sending required production data to the APM tool), or by configuring the APM tool (e.g., to collect operations data for a given percentage of users or only during system load below x percent).

Data filtering controls how the data is filtered for a specific use case. That is, FDD use cases often differ in the types and resolution of required operations data. This includes selecting the type of operations data relevant for the specific FDD use case. For resolution, for instance, performance visualization use cases often require fine-grained data to display accurate dashboards and to allow developers to drill down. Performance prediction, on the other hand, does not require data on the same resolution, and can work on a more coarse-grained sampling.

Data aggregation controls how operations data should be compressed for a use case. For some use cases, basic statistics (e.g., minimum, maximum, arithmetic mean) are sufficient, while other use cases require data on a different level of granularity.

Data integration controls how different types of operations data (or operations data originating from different runtime entities) should be integrated. For instance, in order to calculate the per-request costs of a service, the hourly costs of all virtual machines hosting instances of this service need to be integrated with request counts for this service.

Finally, ***feedback mapping*** links collected, filtered, aggregated, and integrated operations data to development-time source code artifacts. This final step transforms operations data into *feedback*, i.e., data that is immediately actionable for developers.

Each of the first four steps takes the form of transformation functions, taking as input one or more series of operations data $\mathcal{D}$, and produces one or more series of operations data $\mathcal{D}'$ as output. In *feedback specification*, these functions can be represented using, for instance, the complex event processing [Luckham, 2001]

(CEP) abstraction (i.e., using Esper Pattern Language (EPL[5]). The final step, *feedback mapping*, is typically encoded in FDD tools. That is, the knowledge which series of operations data should be mapped to which source code artifacts is generally hard-coded in FDD implementations for a specific use case.

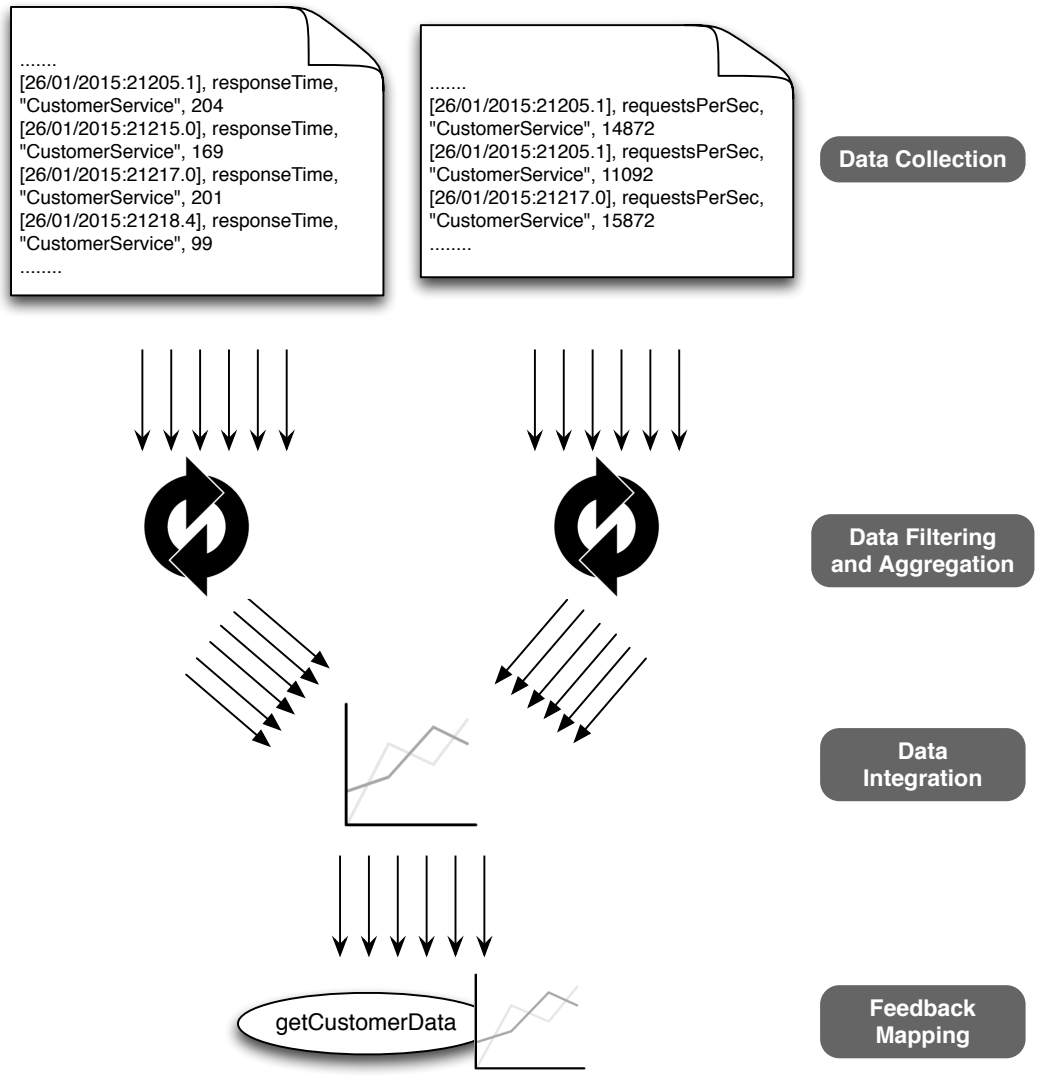


Figure 3.5: Feedback is filtered, aggregated, and integrated operations data, which has been mapped to source code artifacts (e.g., method calls).

[5] `http://www.espertech.com/esper/index.php`

Feedback Freshness

One of the challenges with integrating feedback is identifying when the application has already sufficiently changed so that feedback collected before the change should not be considered anymore (i.e., the feedback became "old" or "stale"). A naive approach would simply ignore all data that had been gathered before any new deployment. However, in a CD process, where the application is sometimes deployed multiple times a day, this would lead to frequent resets of the available feedback. This approach would also ignore external factors that influence feedback, e.g., additional load on a service due to increased external usage.

Hence, we propose the usage of statistical changepoint analysis on feedback to identify whether data should still be considered "fresh". Changepoint analysis deals with the identification of points within a time series where statistical properties change. For the context of observing changes of feedback data, we are looking for a fundamental shift in the underlying probability distribution function. In a time series, we assume that the observations come from one specific distribution initially, but at some point in time, this distribution may change. Recalling the series of observations in operations data in Section 6.4, $\mathcal{D} = \{d_1, d_2, \ldots, d_n\}$, a changepoint is said to occur within this set when there exists a point in time, $\tau \in \{1, ..., n-1\}$, such that the statistical properties of $\{d_1, ..., d_\tau\}$ and $\{d_{\tau+1}, ..., d_n\}$ exhibit differences. The detection of these partition points in time generally takes the form of hypothesis testing. The null hypothesis, H_0, represents no changepoint and the alternative hypothesis, H_1, represents existing changepoints. In previous work, we have already shown that changepoint analysis can be successfully employed to detect significant changes in the evolution of operations data [Cito et al., 2014b, Cito et al., 2015a].

3.4.3 Supporting Developers With Feedback

The purpose of feedback is thus by definition to support developers in reasoning about the future behavior of the systems they create. We can distinguish two types of FDD use cases: analytic FDD and predictive FDD. The former

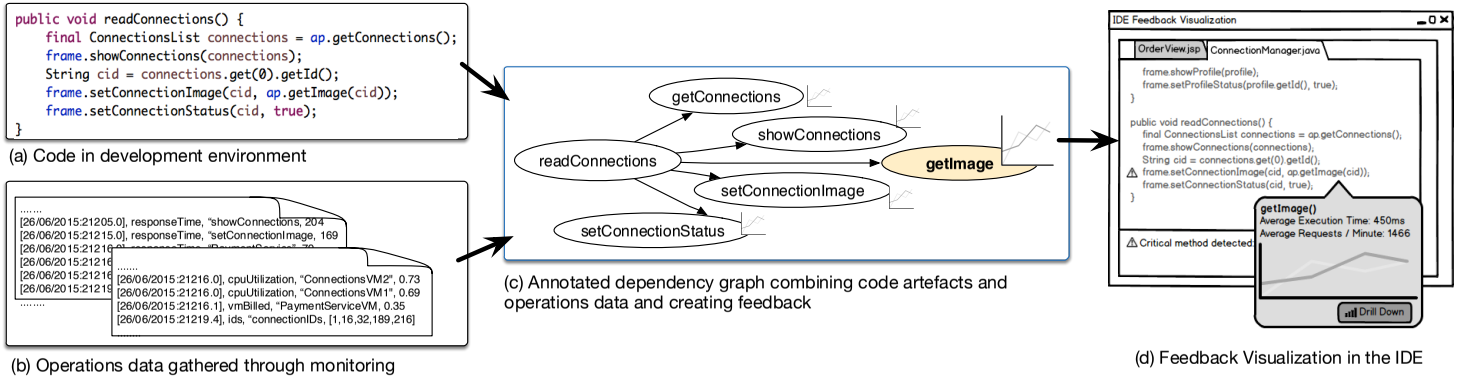


Figure 3.6: How visualization works: (a) A developer examines the method *readConnections*. (b) The application is in production and operations data is available from concrete traces. (c) The dependency tree is annotated with operations data, creating feedback. (d) The feedback visualization is integrated in the code view of the IDE.

deals with analyzing operations data and its relation to development artifacts *a-posteriori*. The latter provides a *prediction* of future system behavior under certain assumptions. In practice, analytic FDD often takes the form of feedback visualization, while predictive FDD is primarily concerned with inferring (as of yet unknown) operations data prior to deployment of a change.

Feedback Visualization

After feedback data is collected it needs to be displayed to the developer in a meaningful context to become actionable. As with all visualizations, the chosen visualization techniques should be appropriate for the data at hand, allowing the user to interactively explore the recorded feedback data and quickly identify patterns. In the context of FDD, the more interesting challenge is to put the implicitly existing link between feedback data and the development-time artifacts (*feedback mapping*) to good use. Figure 3.6 illustrates the process of feedback visualization starting from the developer's code view and execution traces to how standard IDE views are being enriched with feedback.

A developer is examining `readConnections()`, consisting of a small number of method calls. Different kinds of operations data on these methods have been made available by monitoring solutions. A combination of simple static analysis (extracting a simplified AST and call graph in Figure 3.6a) and dynamic analysis (extracting relevant metrics from concrete traces in Figure 3.6b) results in the annotated dependency graph seen in Figure 3.6c. Note that the `getImage` node is highlighted, as it is the only artifact deemed relevant in this scenario. Relevance is determined from a combination of parameters of *feedback control* and statistics calculated from the values of the attached feedback data (e.g., methods with an average execution time over a developer-defined threshold). These artifacts are then highlighted in the IDE through warnings and overlays in the exact spot the identified issue occurred, as depicted in a mockup in Figure 3.6d. Developers are now able to utilize this targeted feedback to guide their design decisions and improve the application. Exemplary concrete visualization techniques that we have experimented with in our concrete tooling are shown in Section 3.5.

Predicting Future Behavior Based on Feedback

Predictive FDD aims at deriving the impact of changes during development in an application based on existing feedback. Figure 6.2 illustrates the steps leading to the prediction of future behavior of an application. A developer changes the code of the method `overallRating()`, adding an iteration over suppliers (`getSuppliers()`) and a call to a different service (`getPurchaseRating()`). Figure 6.2b shows how this code change transforms the dependency graph (described in Section 6.4). The change introduced 3 new nodes where feedback is already available from existing traces: (1) `getSuppliers`, (2) collection size of `suppliers`, and (3) `getPurchaseRating`. The steps "Statistical Inference" and "Feedback Propagation", illustrated in 6.2c, complete the prediction cycle. Feedback for the iteration node `Loop:suppliers` is inferred using the feedback of its child nodes (`size:suppliers` and `getPurchaseRating`) as parameters for a statistical inference model, identifying it as a *critical entity*. The concrete statistical model is specific to the use case. For instance, for one of our use cases (*PerformanceHat* in Section 3.5.2), we chose to implement a quite simplistic model for loop prediction. We model the total execution time of a loop, $\tau(l)$, as the sum of the average execution times, $\bar{\tau}$, of all methods within the loop, $\{l_{m,1}, \ldots, l_{m,n}\}$, times the average collection size, $|l|$, the loop is iterating over: $\tau(l) = \sum_{i=1}^{n} \bar{\tau}(l_{m,i}) \times |l|$. Depending on the specific application nature, the complexity of these inference models can vary greatly. In a last step, all derived feedback from changes are propagated in the nodes of the graph following its dependency edges. This kind of prediction allows us to warn developers about possible issues of their code changes prior to even running the application in production, as shown in a mockup in Figure 6.2d.

3.5 Case Studies

As discussed in Section 3.4, the abstract idea of FDD can be instantiated in various ways, and through various concrete developer tools. We now discuss three concrete tools which implement the FDD paradigm in different ways.

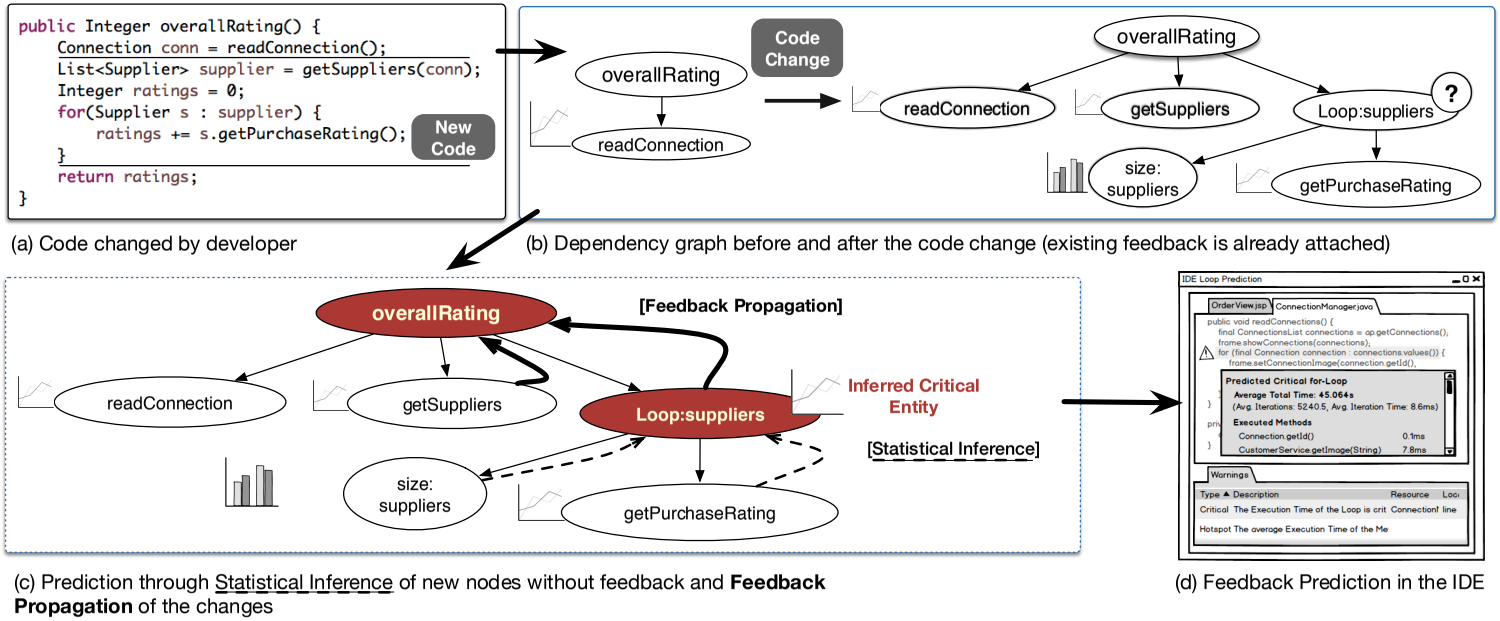


Figure 3.7: How Prediction works: (a) A developer changes the program and adds a loop within the method `overallRating()`. (b) The annotated dependency graph changes accordingly with the change, adding the loop (Loop:suppliers) and its nodes (*size:suppliers* and *`getPurchaseRating`*). (c) Prediction and change impact analysis is achieved through statistical inference and feedback propagation. (d) The prediction is integrated in the code view of the IDE, warning the developer about a preformance-critical change.

3.5.1 FDD-Based Expensive Artifacts Detection

Performance Spotter is an experimental set of tools developed on top of the SAP HANA Cloud Platform[6]. The aim of *Performance Spotter* is to help the developers in finding expensive development artifacts. This has been achieved by creating analytic feedback based on collected operations data (see Section 3.4.2) and mapping this feedback onto corresponding development artifacts in the IDE. Other information such as the number of calls and the average execution time are derived by aggregating the collected data. *Performance Spotter* provides ways to promote performance awareness, root cause analysis, and performance analysis for external services. Hence, *Performance Spotter* is one instantiation of an analytic FDD tool.

Performance Awareness. *Performance Spotter* helps developers to become aware of potential performance issues by observing, aggregating and then visualizing the collected metrics of artifacts. Figure 3.8, illustrates *Performance Spotter*'s Functions Performance Overview. On the left side of the figure, a Javascript code fragment is depicted. On the top right, a list of functions with their relative execution times to other functions is visualized. The blue bars represent the relative execution times. On the bottom right, a diagram illustrates the average execution times of selected functions over time. We can identify poorly performing functions by using the given overview. For instance, Figure 3.8 shows that the average execution time of `getPOs()` is very large in relation to other functions, and that the performance of this function has recently decreased, Furthermore, execution times have increased significantly at particular application runs (*problematic sessions*) and stayed almost stable afterwards. Identifying problematic sessions enables developers to analyze the impact of code as well as resource changes on certain artifacts' performance behavior. In this particular case, the cause of increasing the execution times was code changes in `getPOItems()` before each problematic sessions. However, discovering the root cause of the problem required more insight into the collected monitoring data.

[6] http://hcp.sap.com

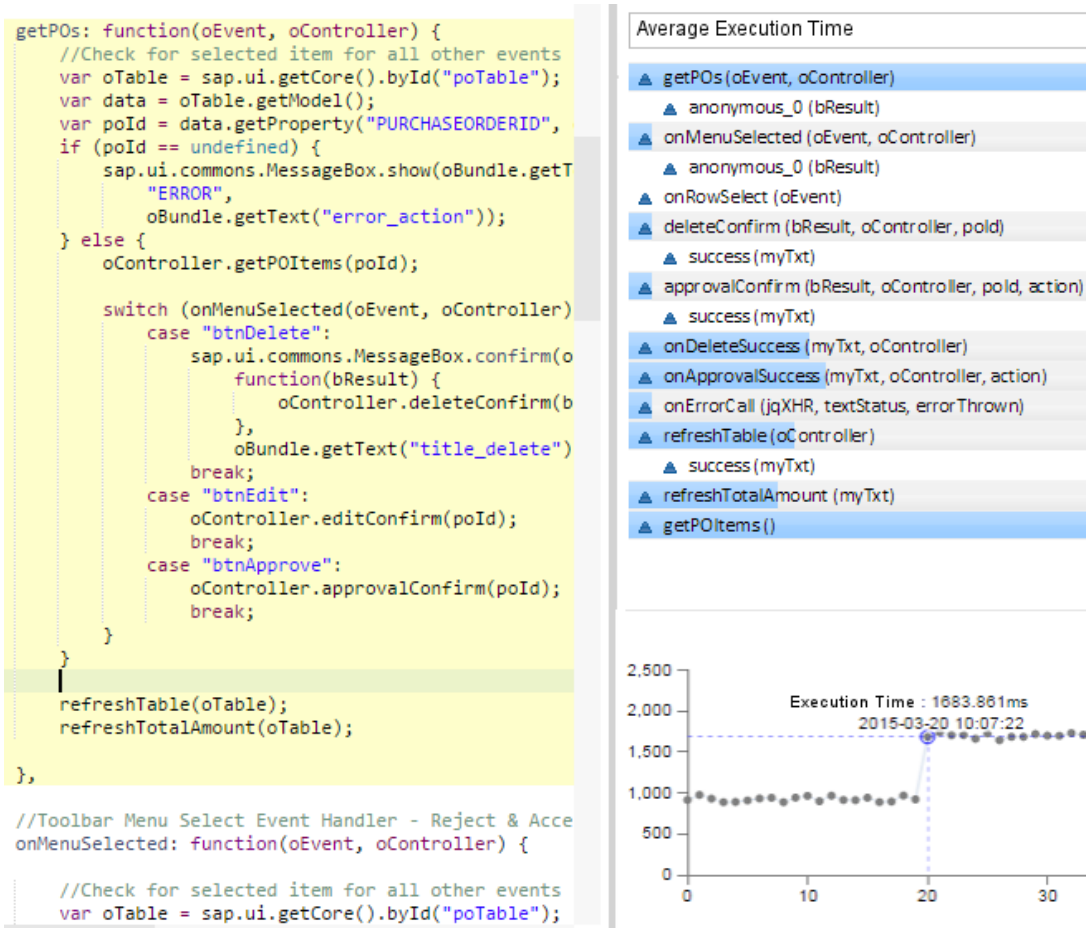


Figure 3.8: *Performance Spotter*'s Functions Performance Overview.

Root Cause Analysis. However, knowing that a development artifact suffers from poor performance is by itself not sufficient. Developers need to find the root cause of such issues. Thus, another feature of *Performance Spotter* is Functions Flow, which builds an annotated dependency graph of functions. The nodes of this graph are function calls and there is an edge between two nodes if one function calls another (i.e., a call graph). The nodes are annotated with relevant feedback (e.g., execution time). Having a visualization of the dependency graph of functions, we can find the root cause of performance issues by traversing the graph and following the poorly performing nodes. The Functions Flow of the artifact `getPOs()` is depicted in Figure 3.9, showing that `getPOItems()` is

the most expensive function call, i.e., it is the root cause of the performance problem.

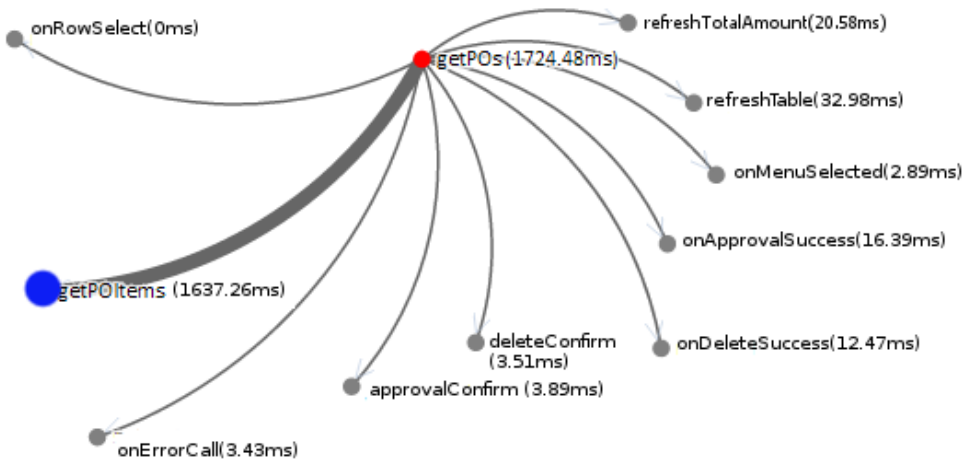


Figure 3.9: *Performance Spotter*'s Functions Flow helps the developer to find expensive functions.

External Service Performance Analysis. Since in many cases an external service (e.g., database) is called within an application, it is necessary to keep track of its behavior from an application's perspective. Previously, we have detected `getPOItems()` as the cause for the high execution time of `getPOs()`. Internally, this method uses a number of database calls, which indicates that improving the database calls would improve the overall performance of the function. *Performance Spotter* provides a feature to analyze the database statements directly from where they are called. Figure 3.10 shows a piece of code (on the top) and the Database Performance Overview (on the bottom). This feature enables the developer to find the expensive database statements by sorting and/or filtering them.

```
256         query = 'SELECT SUM(ITEM.TOTALAMOUNT) from "Purchase.Header" HEADER '
257                 + ' INNER JOIN Purchase.Item ITEM on (ITEM.PURCHASEORDERID = HEADER.PURCHASEORDE
258                 + ' where HEADER.PURCHASEORDERID = ?';
259
260       pstmt = conn.prepareStatement(query);
261       pstmt.setString(1, purchaseOrderID);
262       rs = pstmt.executeQuery();
263
264       processTotalAmount(rs);
265
266         query = 'SELECT HEADER.APPROVALSTATUS'
267                 + ' from "Purchase.Header" HEADER '
268                 + ' where HEADER.PURCHASEORDERID = ?';
269
270       pstmt = conn.prepareStatement(query);
271       pstmt.setString(1, purchaseOrderID);
272       rs = pstmt.executeQuery();
273
274       processApprovalStatus(rs);
275
276         query = 'SELECT HEADER.CONFIRMSTATUS'
277                 + ' from "Purchase.Header" HEADER '
278
```

Database Performance Overview

Execution Step	Statement	Average Execution Time	Number of Calls
9	SELECT SUM(ITEM.TOTALAM...	11.8 ms	237
14	SELECT HEADER.APPROVAL...	2.7 ms	158
19	SELECT HEADER.CONFIRMS...	2.0 ms	215

Figure 3.10: *Performance Spotter*'s Database Performance Overview helps the developer to find expensive database calls.

3.5.2 FDD-Based Prediction of Performance Problems

A second concrete implementation of the FDD paradigm is *PerformanceHat*, a prototypical Eclipse IDE plugin that is able to predict performance problems of applications during development in the IDE (i.e., prior to deployment). It works with any underlying (cloud) platform, as long as the required operations data (see discussion below) is available. *PerformanceHat* is consciously designed in a way that it can easily interface with a variety of monitoring solutions as a backend for operations data (e.g., APM solutions such as NewRelic). The current proof-of-concept version of *PerformanceHat* provides two main features, *hotspot detection* and *critical loop prediction*.

Detecting Hotspots. Hotspots refer to methods that, in the production execution of the application, make up a substantial part of the total execution time of the application (cf., "expensive artifacts" in the previous case study discussion). In a traditional environment this information could be looked up in dashboards of APM solutions, requiring a context switch from the development environment to the operations context of performance dashboards. It also requires further navigation to a specific location in these dashboards. In the FDD paradigm, such hotspot methods are reported as warnings attached to a software artifact within the development environment. Figure 3.11 gives an example of a hotspot. Notice that hotspot methods are identified both at method definition level (e.g., `private void login()`) and method call level (e.g., `Datamanager(b).start()`) in Figure 3.11. When hovering over the annotated entities a tooltip displays summary statistics (currently the average execution time) as a first indicator of the performance problem, as well as a deep link to a dashboard visualizing the time series of operational metrics that led to the feedback.

For hotspot detection, *PerformanceHat* requires only method-level response times in `ms`, as delivered by state-of-the-art monitoring solutions. For program statements for which no response times are available, a response time of `0 ms` is assumed. In practice this means that, oftentimes, we primarily detect hotspots of statements that implement interactions with external components or services (e.g., database queries, remote method invocations, JMS messaging, or invocations of REST interfaces). In Figure 3.11, `login()` is identified as a hotspot, as `DataManager.start(..)` invokes an external REST service in the Microservice based architecture of the *MicoSRM* illustrative example (see Section 6.3). Other statements, for instance `b.getPassword()`, are ignored as the response times for these statements are negligible.

Critical Loop Prediction.

Performance problems are often related to expensive loops [Jin et al., 2012]. As described in Section 3.4.3, in FDD we predict the outcome of changes in code by utilizing existing data to infer feedback for new software artifacts. In *PerformanceHat* we are able to do so for newly introduced loops over collections (i.e., *foreach* loops). In the initial stages of our prototype we propose a simplified

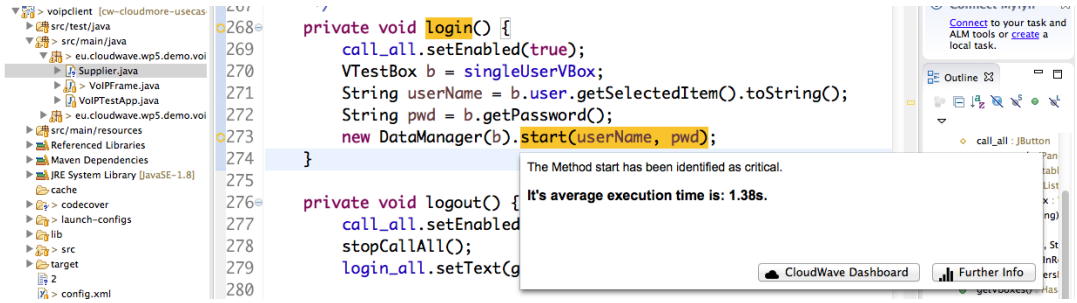


Figure 3.11: The *PerformanceHat* plugin warning the developer about a "Hotspot" method.

model over operations data on collection sizes and execution times of methods to infer the estimated execution time of the new loop (as discussed in Section 3.4.3). Figure 3.12 gives an example of a so-called *Critical Loop*. When hovering over the annotated loop header a tooltip displays the estimated outcome (*Average Total Time*), the feedback parameters leading to this estimation (*Average Iterations* and *Average Time per Iteration*), and the execution times of all methods in the loop. This information enables developers to dig further into the performance problem, identify bottlenecks and restructure their solutions to avoid poor performance even *before* committing and deploying their changes.

We are in the process of releasing *PerformanceHat* as an open source project at GitHub[7], as a plugin compatible with Eclipse Luna (Version 4.4.1) and onwards. A screencast demonstrating *PerformanceHat*'s capabilities can be found online[8] as well.

3.5.3 FDD-Based Prediction of Costs of Code Changes

It is often assumed that deploying applications to public, on-demand cloud services, such as Amazon EC2 or Google AppEngine, allows software developers to keep a closer tab on the *operational costs* of providing the application. However, in a recent study, we have seen that costs are still usually intangible to software developers in their daily work [Cito et al., 2015b]. To increase awareness of

[7]http://www.github.com/sealuzh/PerformanceHat
[8]http://bit.ly/PerformanceHat

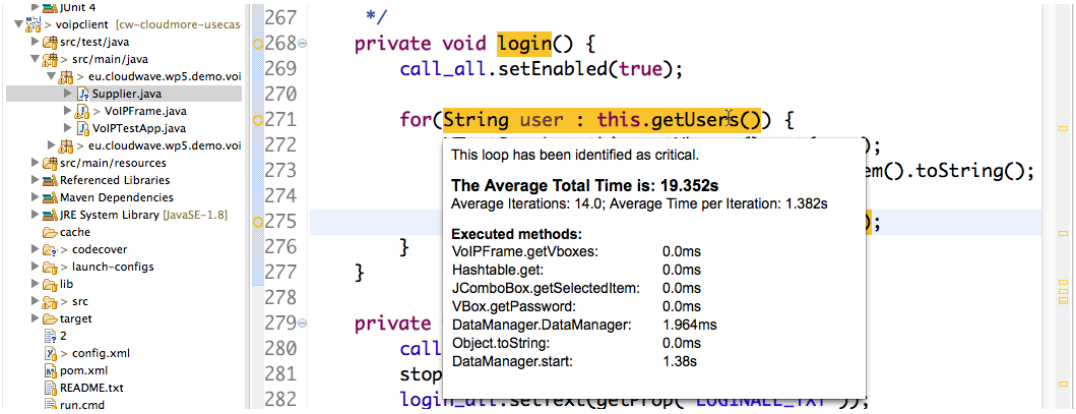


Figure 3.12: Prediction of Critical Loops in the *PerformanceHat* plugin.

how even small design decisions influence overall costs in cloud applications, we propose integrated tooling to predict costs of code changes following the FDD paradigm. We present our ideas on how we can predict costs *induced by introducing a new service* and by *replacing existing services*. Unlike the *Performance Spotter* and *PerformanceHat* use cases, our work on this idea is still in an early stage. We are currently in the process of building a proof-of-concept implementation for these ideas on top of Eclipse, NewRelic and AWS CloudWatch.

Costs through new Services. When considering services deployed on cloud infrastructure, increasing load usually leads to the addition of compute instances to scale horizontally or vertically. In this scenario, illustrated in Figure 3.13, we introduce a new service *SupplierRatingService* that invokes the existing *PurchaseOrderService*. The IDE tooling provides information on the deployment and cost structure of the existing service (*StatusQuo*) and provides an impact analysis on how this structure would change (*Expected Impact*) based on load pattern parameters (*Incoming Requests*). The load pattern parameters would be estimated by leveraging monitoring data for similar services in the application and can be adjusted by the developer to perform sensitivity analysis.

Costs induced by Replacing Services. Another, similar, scenario in Figure 3.14 considers the replacement of an invocation within an existing service (*OtherPaymentService*) with a new service (*NewPaymentService*). In this case, the

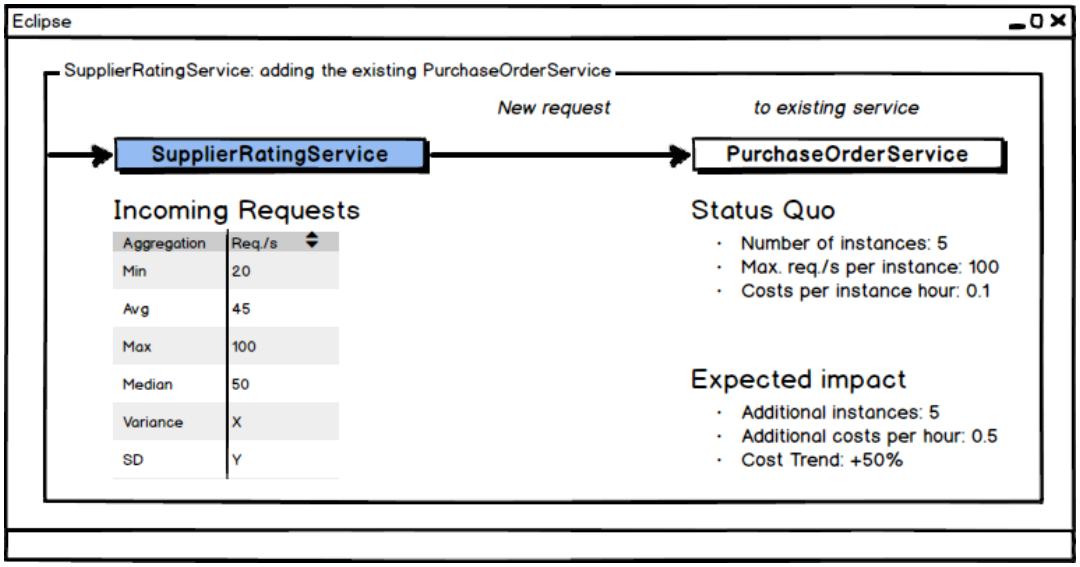


Figure 3.13: IDE Tooling displaying how the new service SupplierRating has an impact on the cost of existing service of the PurchaseOrder.

load pattern parameters are known and pre-populated in the interface (*Incoming Requests*). The impact analysis (*Expected Impact*) differs from the previous case in that the model takes into account partial rollouts (*Simulation*). This allows for more complex sensitivity analysis scenarios.

3.6 Challenges Ahead

Based on our case study implementations, as well as based on initial experiments and discussions with practitioners, we have identified a small number of interesting challenges that need to be addressed before the FDD idea can be deployed on a larger scale.

Data Access and Privacy. The availability of rich, up-to-date, and correct operations data is the cornerstone upon which FDD builds. While modern cloud and APM solutions already provide a wealth of data, we have still encountered concerns regarding the availability of some of the data discussed in Figure 3.4. Most importantly, production data (e.g., user information, order data) will,

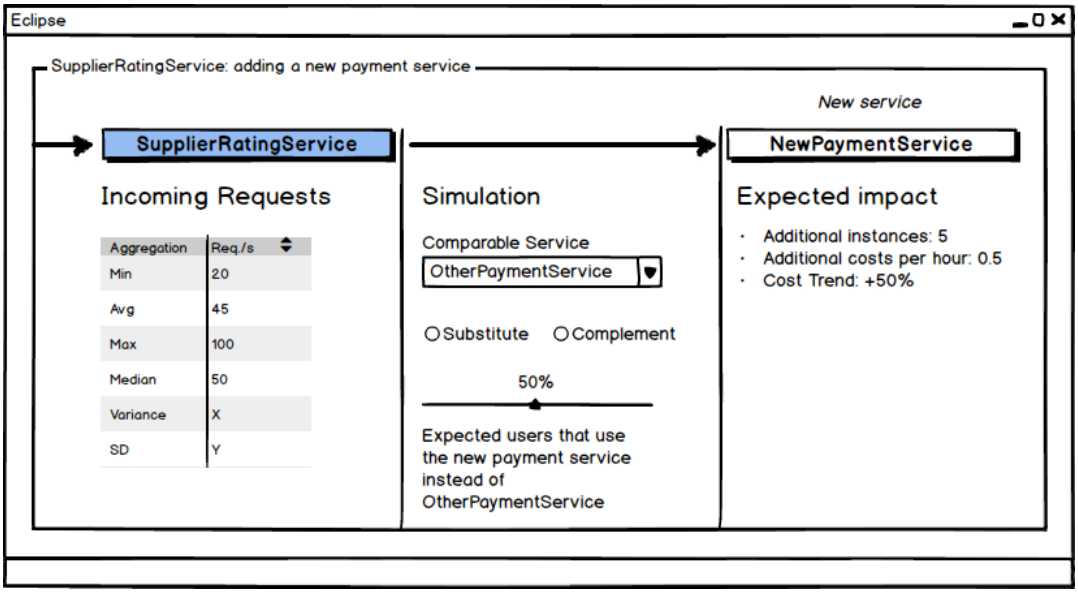


Figure 3.14: IDE Tooling displaying how the replacement of an existing service (OtherPaymentService) by a new service (NewPaymentService) has an impact on overall costs.

in many cases, be unavailable to engineers due to privacy and data protection concerns. Consequently, our initial use cases in *PerformanceHat* and *Performance Spotter* did not make use of these types of operations data. Relatedly, when deploying the FDD idea on Web scale, we will also face the orthogonal problem that there will in many cases actually be *too much* data available to use directly in developer-facing FDD tools.

For both cases, it will become necessary to invest more research into how operations data is actually harvested in production. In terms of privacy protection, we envision future monitoring solutions to be privacy-aware, and be able to automatically anonymize and aggregate data to the extent required by local data protection laws or customer contracts. We expect that many lessons learned from privacy-preserving data mining [Verykios et al., 2004] can be adapted to this challenge. Further, we need to devise monitoring solutions that are able to sample operations data directly at the source. While it is easy to only generate data for a subset of calls (e.g., only track 1% of all service invocations), doing so while

preserving the relevant statistical characteristics of the underlying distribution is not trivial. However, when taking elasticity of the cloud into consideration (e.g., horizontal scaling), we are able to trade higher monitoring coverage (having the same performance) with higher costs due to additional resources needed for instrumentation.

Confounding Factors. All prototypical FDD use cases discussed in Section 3.5 are operating on the simplified assumption that feedback is largely free from confounding factors, such as varying service performance due to external influences (e.g., variance in networking performance, external stress on the service, or a service scaling up or down due to changes in customer behavior). This problem is amplified by the fact that cloud infrastructures are known to provide rather unpredictable performance levels [Leitner and Cito, 2014]. When deploying FDD in real projects, such confounding factors will lead to false positives or negatives in predictions, or produce misleading visualizations.

More work will be required on robust data aggregation methods that are able to produce useful feedback from noisy data. These methods will likely not only be statistical, but integrate heterogeneous operations data from different levels in the cloud stack (e.g., performance data, load data, scaling information) to produce clearer and more accurate feedback. First steps in this direction have already been conducted under the moniker 3-D monitoring [Marquezan et al., 2014].

Information Overload. Another challenge that all FDD use cases discussed in Section 3.5 face is how to display the, and only the, information that is relevant to a given developer. Ultimately, visualizations and predictions generated by FDD tools need to not only be accurate, but also be relevant to the developer. Otherwise, there is a danger that developers start ignoring or turning off FDD warnings. This problem is amplified by the fact that different developers and project roles care about different kinds of problems.

A technical solution to this challenge is customization support. Feedback control (see Section 3.4.2) enables developers to turn specific feedback on and off, to restrict feedback to certain services or methods, or to change the thresholds for warnings. For instance, in *PerformanceHat*, developers can change the threshold

from which a method is displayed as a hotspot, either globally or on a per-method level. However, in addition, more research is necessary to know which kinds of feedback, warnings and predictions are typically useful for which kinds of developers and projects. Existing research in software engineering on information needs [Breu et al., 2010,Fritz et al., 2010] and developer profiles [Brandtner et al., 2015] can serve as a basis for this work.

Costs as Opportunity. Our aim to integrate monetary considerations within the FDD paradigm is to increase awareness about the costs design decisions have when developing for the cloud. However, this awareness should go beyond considering costs solely as a burden, but rather as an opportunity (e.g., better performance through more computing power leads to more sales).

Possible solutions to this challenge include associating cost models in FDD with business metrics, as well as proper information visualization to underline opportunities.

Technical Challenges. Finally, there are a number of technical challenges that need to be tackled when realizing the FDD paradigm in industry-strength tools. Specifically, in our work on *PerformanceHat* and SAP HANA's *PerformanceSpotter*, we have seen that implementing FDD in a light-weight way, such that the additional static analysis, statistical data processing, and chart generation does not overly slow down the compilation process or the perceived responsiveness of the IDE, is not trivial from a technical perspective. *Feedback control*, as discussed in Section 3.4.2, mitigates this challenge somewhat by minimizing the nodes in the generated dependency graphs. However, particularly prediction of the impact of code changes (see Section 3.4.3) is currently taxing in terms of performance, as predicted changes to the feedback associated to one method need to be propagated through the entire application AST. We are currently investigating heuristics to speed up this process.

3.7 Related Work

A substantial body of research exists in the general area of cloud computing [Heilig and Voss, 2014], including work on cloud performance management and improve-

ment (e.g., [Iosup et al., 2011, Gambi and Toffetti, 2012, Shieh et al., 2010], among many others). However, as [Barker et al., 2014] notes, software development aspects of cloud computing and SaaS are currently not widely explored. We have recently provided a first scientific study that aims to empirically survey this field [Cito et al., 2015].

One of the observations of this study was that cloud platforms and APM tools (e.g., the commercial solutions NewRelic or Ruxit[9]) make a bulk of data available for software developers, but that developers currently struggle to integrate the provided data into their daily routine. Existing research work, for instance in the area of service [Liu et al., 2004, Rosenberg et al., 2006, Michlmayr et al., 2009], cloud [Meng et al., 2011, Marquezan et al., 2014], or application monitoring [van Hoorn et al., 2012], provides valuable input on how monitoring data should be generated, sampled, and analyzed. There is very little research on how software developers actually make use of this data to improve programs. FDD is an approach to address this gap. However, FDD is not a replacement of APM, but rather an extension that makes use of the data produced by APM tools. To this end, our work bears some similarities to research in the area of live trace visualization [Greevy et al., 2006, Fittkau et al., 2013]. However, our work goes beyond the scope of visualization of program flow. Some parts of FDD, specifically the use cases more geared towards predicting performance problems before deployment, are based on prior work in the area of performance anti-patterns [Smith and Williams, 2000, Wert et al., 2014] and their automated detection. Our implementation of these predictive FDD use cases also leans heavily on prior work in static and, especially, in dynamic program analysis [Park, 1993]. It is possible to view FDD as a pragmatic approach to bring cloud development closer to the idealistic view of *live programming* [McDirmid, 2007, McDirmid, 2013]. Live programming aims to entirely abolish the conceptual separation between editing and executing code. Developers are editing the running program, and immediately see the impact of any code change (e.g., on the output that the program produces). Hence, developers can make use of immediate feedback to steer the next editing steps [Burckhardt et al., 2013].

[9] https://ruxit.com

Existing work on live programming has also stressed the importance of usable IDEs and development environments [Lemma and Lanza, 2013]. FDD is an approach to take similar core ideas (bringing feedback, e.g., in terms of execution time, back into the IDE), but plugging them on top of existing programming languages, tools and processes, rather than requiring relatively fundamental changes to how software is built, deployed, and used. FDD also acknowledges that, for enterprise-level SaaS applications, local execution (as used in live programming) is not necessarily a good approximation for how the application will perform in a Web-scale production environment. Hence, the production feedback used in FDD is arguably also of more use to the developer than the immediate feedback used in live programming.

3.8 Conclusions

In this paper, we presented our vision on Feedback-Driven-Development, a new paradigm that aims at seamlessly integrating feedback gathered from runtime entities into the daily workflow of software developers, by associating feedback to code artifacts in the IDE. We discussed how operations data produced by APM solutions is filtered, aggregated, integrated, and mapped to relevant development-time artifacts. This leads to annotated dependency trees, which can then be utilized for different use cases. In this paper, we have focused on two types of FDD use cases. Analytic FDD focuses on displaying relevant collected feedback visually close to the artifacts that it is relevant for, hence, making the feedback actionable for developers. Predictive FDD makes use of already collected feedback in combination with static analysis to predict the impact of code changes in production. We have exemplified these concepts based on two implementations, on top of Eclipse as well as on top of SAP's HANA cloud solution.

We believe that the general idea of FDD will, in the upcoming years, become more and more important for software developers, tool makers, and cloud providers. Currently, we are arguably only seeing the tip of the iceberg of possibilities enabled by integrating operations data into the software development process and tools. However, we also see a number of challenges that need to be

addressed before the full potential of FDD can be unlocked. Primarily, we need to address questions of privacy and correctness of data. Further, more academic studies will be required to identify which kinds of feedback developers actually require in which situations. Finally, our practical implementations have also shown that there are numerous technical challenges to be addressed when trying to make Web-scale data useful for the developer in (or close to) real-time.

4

PerformanceHat Augmenting Source Code with Runtime Performance Traces in the IDE

Jürgen Cito *Philipp Leitner, Christian Bosshard, Markus Knecht, Genc Mazlami, Harald C. Gall*

Submitted to the 40th ACM/IEEE International Conference on Software Engineering, Demonstration Track (ICSE Demonstrations) 2018

Contribution: Framework design, prototype implementation, and paper writing

Abstract

Performance problems observed in production environments that have their origin in program code are immensely hard to localize and prevent. Data that can

help solve such problems is usually found in external dashboards and is thus not integrated into the software development process. We propose an approach that augments source code with runtime traces to tightly integrate runtime performance traces into developer workflows. Our goal is to create operational awareness of performance problems in developers' code and contextualize this information to tasks they are currently working on. We implemented this approach as an Eclipse IDE plugin for Java applications that is available as an open source project on GitHub. A video of PerformanceHat in action is online: `https://youtu.be/fTBBiylRhag`

4.1 Introduction

Software developers produce high volumes of code, usually in an IDE, to provide functionality that solves problems. Code is then often deployed in complex systems exhibiting distributed architectures and scalable infrastructure. Through observation and instrumentation, we collect logs and metrics (we collectively call them runtime traces) to enable comprehension of the inner workings of our deployed software. Performance problems that occur in production of these complex systems often originate in development [Hamilton, 2007]. Developers then have the challenging task to reason about runtime behavior and determine why certain parts of their code do not exhibit desired performance properties. For some performance problems, this reasoning can be supported by information provided by profilers executed in a local environment. However, there is a whole class of performance anomalies that occur in production that cannot be solved by the information provided by these local profilers. Tracing these class of problems back to their origin in the source code and debugging them requires the inspection of runtime traces from production. Since our deployment is distributed and complex, so are the traces collected at runtime. To mitigate this complexity, a large body of academic [Aceto et al., 2013, van Hoorn et al., 2012] and industrial work, both in open source and commercial solutions attempt to provide insight into runtime behavior by introducing dashboards that visualize

data distributions in the form of time series graphs and enable search capabilities for trace properties.

This particular design and level of abstraction of runtime information is largely designed to support the reactive workflow of software operations, whose responsibilities require them to think in systems, rather than the underlying source code. This traditional view of operations analytics does not fit into the workflow of developers, who struggle to incorporate information from traces into their development and debugging activities. This is in line with existing work that argues for program analysis to be effective, need to be smoothly integrated into the development workflow [Sadowski et al., 2015, Cito et al., 2015b].

PerformanceHat. We argue that the abstraction level at which runtime performance traces are presented in the current state-of-the-art is not well-suited to aid software developers in understanding the operational complexities of their code. In this paper, we present *PerformanceHat*, an open source IDE plugin[1] that tightly integrates runtime performance aspects into the development workflow by augmenting the source code with performance traces collected in production environments. Distributed runtime traces are modeled in a way that they can be visually attached to source code elements (e.g., method calls and definitions, loop headers, collections).

All efforts going into the conception and implementation of our approach are guided by the following goals:

- *Operational Awareness:* By integrating runtime aspects into source code and, thus, into the development workflow, developers become more aware of the operational footprint of their source code.

- *Contextualization:* When runtime performance aspects are visible when working with source code, they are contextualized to the current task of the developer (e.g., feature development, debugging). Thus, developers do not have to establish their current context in external tools through search.

[1] http://sealuzh.github.io/PerformanceHat/

4.2 Approach Overview

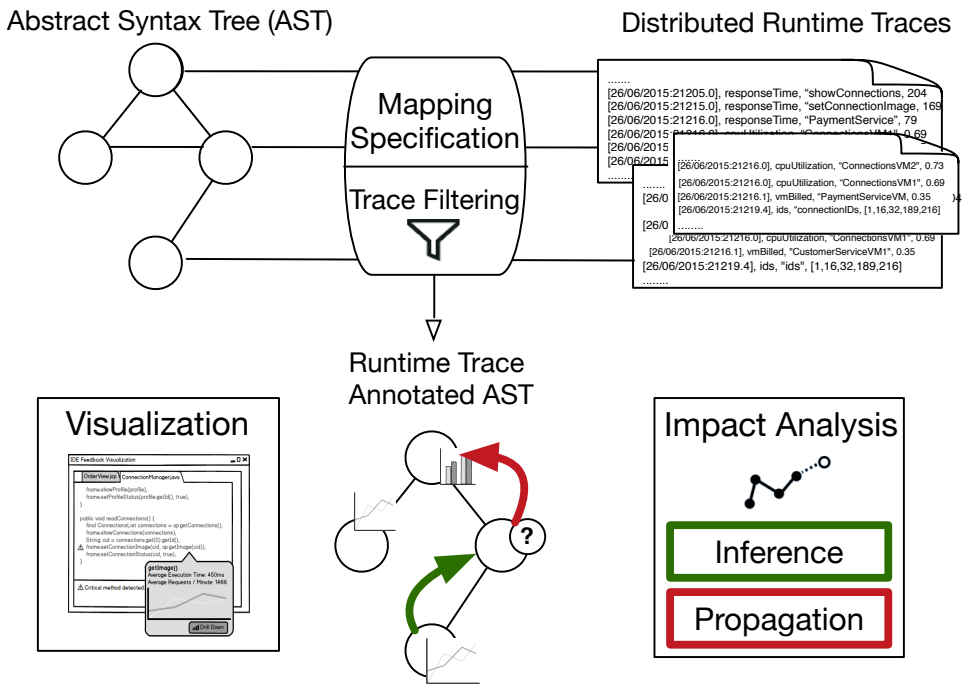
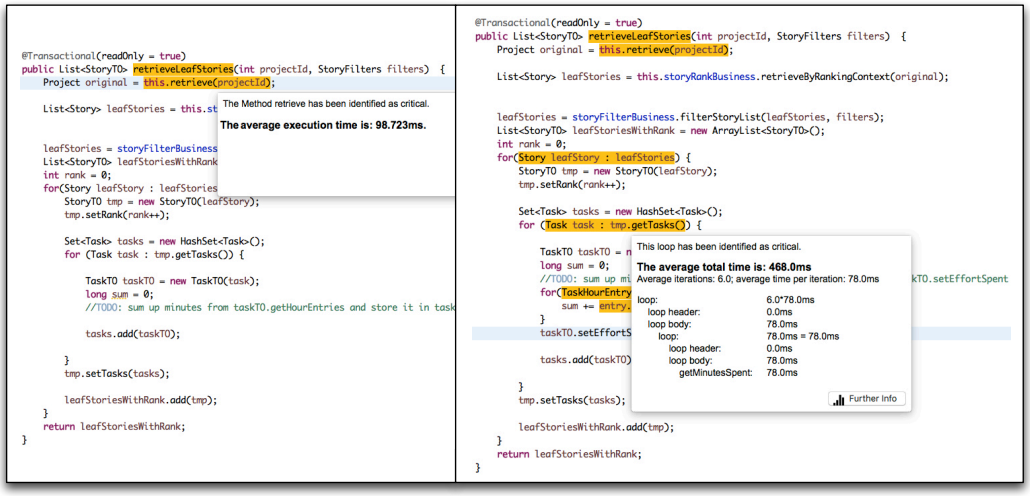


Figure 4.1: Conceptual Framework for the proposed approach. Nodes of source code, represented as a simplified Abstract Syntax Tree (AST), are annotated with runtime performance traces.

The basic theory underlying the approach is that static information available while working on maintenance tasks in software development does not suffice to properly diagnose problems that occur at runtime. The conjecture then is: Augmenting source code with dynamic information from the runtime yields enough information to the software developer to (1) make informed, data-driven decisions on how to perform code changes, (2) perceive code as it is experienced by its users. There are four particular abstractions that highlight the essence of our approach: *specification*, *trace filtering*, *impact analysis*, and *visualization*. We provide an overview in the following, details can be found in the full paper [**?**].



(a) Displaying Operational Footprint (b) Inferring the Performance of Newly-Written Code

Figure 4.2: Our tool *PerformanceHat* as viewed in the development workflow in the Eclipse IDE.

Mapping/Specification In an initial step, the abstract syntax tree (AST) of the source code is combined with the dynamic view of runtime information in a process called specification or feedback mapping [Cito et al., 2015b]. A set of traces can be mapped to different AST node types (e.g., method invocations, loop headers) based on different specification criteria in both node and trace. In the case of *PerformanceHat*, we map response time traces based on the fully-qualified method name in Java, that is both available as part of the AST node and in the trace data. While this is, in part, also done by regular profilers, we allow for more complex specification queries. Specifications define declarative queries about program behavior that establish this mapping relationship. In previous work, we demonstrate how we can establish approximative mappings between arbitrary traces from logs to source code through semantic similarity [Cito et al., 2017]. In the IDE, this results in a performance augmented source code view, that allows developers to examine their code artifacts annotated with performance data from runtime traces from production. Examples for such a mapping go from method calls with execution times, usage statistics for features or collections with size distribution.

Trace Filtering Traces are collected from distributed architectures with scalable instances. Data from these traces often exhibit a multi-modal distribution. Attaching all data that corresponds to a code element from the entire trace dataset might be misleading, as different subsets of the data might originate from multiple sources of distributions. Trace filtering enables sensitivity analysis of potential problematic properties of traces (e.g., from multiple data centers, or from users accessing data stores with different characteristics). Further, there are also other reasons to filter certain data points before displaying them to software developers (i.e., data cleaning). This can either be removing measurement errors through fixed thresholds to more complex filtering with interquartile ranges or dynamic filtering adjusted based on trends and seasonality.

Impact Analysis When working on maintenance tasks, software developers often add or delete code. Nodes in the AST of newly added code does not yet have runtime traces that can be mapped. The goals of impact analysis are two-fold: (1) attempt to predict properties of unknown nodes (*inference*), and (2) given new information from inference, update the information of all dependent nodes (*propagation*). A prediction model is given the delta between the annotated AST, with all its attached operational data, and the current AST that include the new changes as parameters to infer new information about the unknown nodes in the current AST. The prediction can be kept as simple as a “back-of-the-envelope" additive model that sums up all execution times within a method, to more complex machine learning models taking into account different load patterns to estimate latency of methods. Impact analysis gives early feedback and gives developers an outlook to the future of their code changes prior to running the application in production.

Visualization Eventually, the trace data that was either mapped or inferred needs to be displayed to the developer in a meaningful context to become actionable. There are two dimensions to the visualization part: (1) how trace data and predictions are displayed, and (2) how developers are made aware that data for certain elements of source code exists in the first place. From previous

steps in this framework, we are usually given a set of data points (distribution), rather than just a point estimate. A possible way to present runtime performance data could be to show a summary statistic as initial information and allow for more interaction with the data in a more detailed view. In terms of diverting attention that data exists, source code elements should be in some way highlighted in the IDE through warnings and overlays in the exact spot of the identified issue.

4.2.1 Implementation

We implemented an instantiation of the described framework as a proof-of-concept for runtime performance traces as a tight combination of components: An Eclipse IDE plugin for Java programs and local server component with a database (*feedback handler*) that deal with storing a local model of trace information and filtering. Performance traces (e.g., execution times, CPU utilization) are attached to method definition and method calls. Impact analysis is currently supported for adding method calls and loops within a method body. Figure 4.2 provides a screenshot of *PerformanceHat* for two basic scenarios. The tool is open source and available on GitHub (including detailed setup instructions)[2].

We depict a high-level architecture that enables scalable feedback of runtime traces in the IDE. In the following, we describe the main components of the architecture (*IDE Integration* and *Feedback Handler*) and discuss some considerations to achieve scalability.

IDE Integration To achieve tight integration into the development workflow, we implemented the frontend of our approach as an IDE plugin in Eclipse for Java. PerformanceHat hooks program analysis, specification/mapping, and inference into Eclipse's incremental builder. This means, whenever a file is saved, we start the analysis for that particular file. Both specification and inference function are designed as extension points and can be embedded by implementing a predefined interface. This allows us to implement different specification queries and inference functions based on the domain and application.

[2] `http://sealuzh.github.io/PerformanceHat/`

For PerformanceHat, we implemented specification queries that map execution times to method definitions and method invocations and, for certain use cases, instrumented collection sizes to for-loop headers. We also implemented inference functions for adding new method invocations and for-loops over collections within existing methods. The plugin handles the interactions between all sub-components and the *feedback handler*.

Feedback Handler The feedback handler component is the interface to the data store holding the feedback in a model that the plugin understands. It is implemented as a Java application, exposing a REST API, with a MongoDB data store. Conceptually, we can think of two separate feedback handlers: (1) deployed feedback handler (remote), and (2) local feedback handler (on the developer's workstation). The deployed feedback handler is installed on the remote infrastructure, close to the deployed system and has an interface to receive or pull runtime information from monitoring systems. The local feedback handler runs as a separate process on the software developer's local workstation. The reasoning for this split is that displaying metrics and triggering inference in the IDE requires fast access to feedback. The local feedback handler acts as a replicated data store of potentially many remote traces. However, the implementation of any particular replication method is left to the particular user as requirements for consistency of local data vary between use cases. To achieve trace filtering, developers could register their own filters as MongoDB view expressions.

Scalability Considerations For any kind of program analysis that is hooked into the incremental build process (i.e., it is executed with every file/project save), researchers and developer tool designers need to ensure immediacy of analysis results.

In initial versions of PerformanceHat, every build process triggered fetching new data from the *remote* feedback handler, introducing network latency as a bottleneck. Initial attempts to grow from smaller, toy examples to larger and more realistic applications showed that this simplistic one-to-one adoption of

the conceptual framework does not scale sufficiently. In an iterative process, we arrived at the architecture depicted on a high level in Figure 4.3. We briefly summarize our efforts to enable scalability for runtime trace matching in development workflows:

- *Local Feedback Handler:* To decrease the time required to load all data required for the analysis it is moved as close as possible to the IDE, on the developer's workstation. This increases coordination between deployed and local instances, but is absolutely necessary to avoid interruptions to the developer workflow.
- *Local IDE Cache:* To further reduce the time for loading metrics from the feedback handler, we introduced a local IDE cache, such that each entry must be fetched only once per session. We used an LRU cache from the Guava[3] open source library with a size of maximal 10000 entries and a timeout of 10 minutes (after 10 minutes the cache entry is discarded, however both these entries are configurable in a configuration file). This reduced the build time significantly.
- *Bulk Fetching:* A significant improvement also occurred when, for an empty cache, we first registered all nodes that required information from the feedback handler and then loaded all information into the cache in bulk.

4.3 Scenarios & User Study

Sample Scenarios The screenshots in Figure 4.2 depict two basic scenarios that are further elaborated in the demonstration video:

- (a) *Awareness and Comprehension for Performance Debugging:* We are displaying execution times observed in production, contextualized on method level. The box with initial information is displayed as developers hover over the marker on the "this.retrieve" method invocation. A button "Further info" opens a more detailed view (e.g., time series graph) from an existing dashboard.

[3] `https://github.com/google/guava`

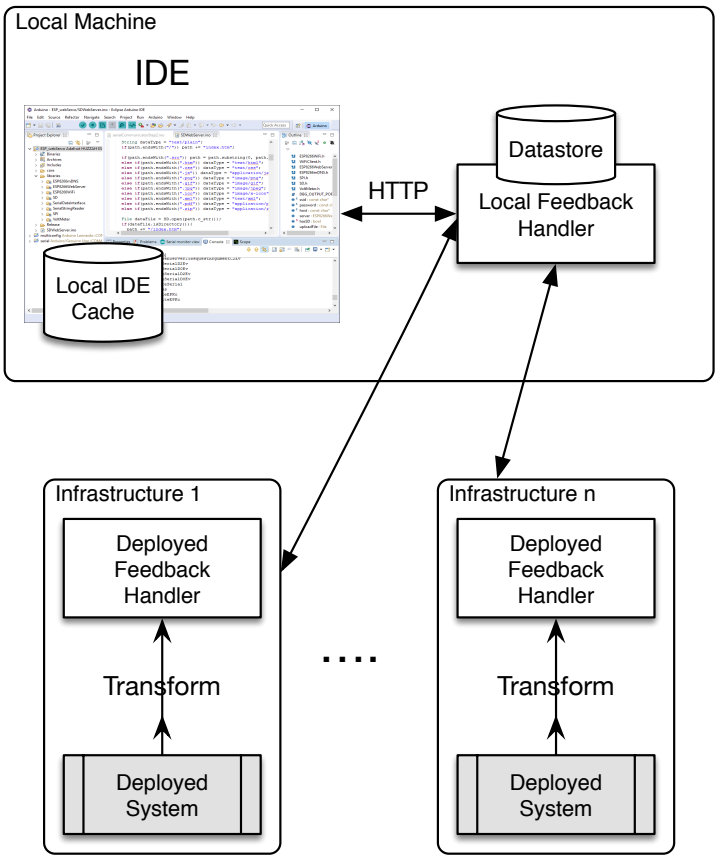


Figure 4.3: Scalable architecture for rapid feedback of runtime performance traces in the IDE

- (b) *Problem Prevention through Impact Analysis:* After introducing a code change, the inference model attempts to predict the newly written code. Further, it is propagated over the blocks of foreach-loops. The box is displayed when hovering over the loop over the Task collection. It displays the prediction in this block and basic supporting information.

User Study Summary We evaluated the effectiveness of PerformanceHat in a controlled user study with 20 professional software developers (as part of a larger study on the approach where we describe our method more rigorously, see Chapter 6). We want to assess whether our approach has an impact of the

time it takes to (1) detect performance problems in maintenance tasks, and (2) find the root cause of these problems. Further, as not all software maintenance tasks potentially introduce a performance problem, we are also curious to learn whether our approach would introduce significant overhead into the development process. To summarize, our study showed that:

- *First Encounter:* Developers were significantly faster in detecting performance problems. We saw one exception to this for one particular task, which after further analysis revealed the limitation of our approach. The task was to introduce a loop into a two nested loops. Our analysis showed that if developers can easily reason about the runtime complexity through statically inspecting the code, our approach does not yield a significant improvement for the *first encounter*.

- *Root-Cause Analysis:* Developers were significantly faster in finding the root-cause of the problem. This even held true for the above mentioned “obvious" performance problem where the first encounter was more easily attainable through code inspection alone. However, the root-cause analysis did require querying performance data to pinpoint the origin of the anomaly.

- *Non-Performance Tasks:* There is a strong indication that our approach does not introduce significant cognitive overhead that “distracts" software developers from regular maintenance tasks (i.e., we were not able to reject the null-hypothesis for these non-performance relevant tasks).

4.4 Conclusion

We presented PerformanceHat, a tool that integrates runtime performance traces into the development workflow by augmenting source code in the Eclipse IDE and providing live performance feedback for newly written code during software maintenance. Its goal is to creates operational awareness of source code and contextualize runtime trace information to tasks in the development workflow. Our empirical results show that PerformanceHat helps software developers detect

and find the root cause of performance problems faster. More details on the study, the conceptual approach, and its related work can be found in Chapter 6.

5

Context-Based Analytics Establishing Explicit Links between Runtime Traces and Source Code

Jürgen Cito, *Fábio Oliveira, Philipp Leitner, Priya Nagpurkar Harald C. Gall*
Published at Proceedings of the 39th ACM/IEEE International Conference on Software Engineering: Software Engineering in Practice (ICSE SEIP) 2017
Contribution: Case study design, data collection, data analysis, prototype implementation, and paper writing

Abstract

Diagnosing problems in large-scale, distributed applications running in cloud environments requires investigating different sources of information to reason about application state at any given time. Typical sources of information available to developers and operators include log statements and other *runtime information* collected by monitors, such as application and system metrics. Just as importantly, developers rely on information related to changes to the source code and configuration files *(program code)* when troubleshooting. This information is generally scattered, and it is up to the troubleshooter to inspect multiple *implicitly-connected* fragments thereof. Currently, different tools need to be used in conjunction, e.g., log aggregation tools, source-code management tools, and runtime-metric dashboards, each requiring different data sources and workflows. Not surprisingly, diagnosing problems is a difficult proposition. In this paper, we propose Context-Based Analytics, an approach that makes the links between runtime information and program-code fragments *explicit* by constructing a graph based on an application-context model. *Implicit* connections between information fragments are *explicitly* represented as edges in the graph. We designed a framework for expressing application-context models and implemented a prototype. Further, we instantiated our prototype framework with an application-context model for two real cloud applications, one from IBM and another from a major telecommunications provider. We applied context-based analytics to diagnose two issues taken from the issue tracker of the IBM application and found that our approach reduced the effort of diagnosing these issues. In particular, context-based analytics decreased the number of required analysis steps by 48% and the number of needed inspected traces by 40% on average as compared to a standard diagnosis approach.

5.1 Introduction

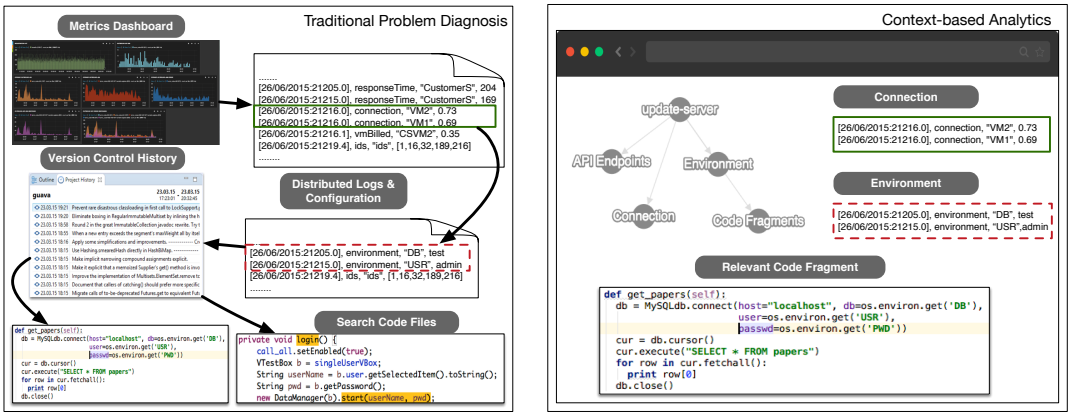
The scalable and ephemeral nature of infrastructure in cloud software development [Cito et al., 2015b] makes it crucial to constantly monitor applications to

gain insight into their runtime behavior. Data collection agents send application and infrastructure logs and metrics from cloud guests and hosts to a centralized storage from which all data can be searched and a variety of dashboards are populated.

The ever-increasing need for rapidly delivering code changes to satisfy new requirements and to survive in a highly-competitive, software-driven market has been fueling the adoption of DevOps practices [Bass et al., 2015] by many companies. By breaking the well-known barrier between development and operations teams, the cultural changes, methodologies, and automation tools brought about by the DevOps phenomenon allow teams to continuously deploy new code to production in cloud environments. It is not uncommon for many companies to deploy new code several times per day [Schermann et al., 2016].

However, when a cloud application faces a problem in production, causing a partial or total outage, this fast code-delivery cycle is suddenly halted. Paradoxically, this extreme agility in deploying new code could potentially slow down the continuous delivery cycle, as problems might happen more often, and take longer to resolve, the faster new code is deployed. Hence, it is paramount that developers and operators are empowered to quickly determine the root cause of problems and fix them. Diagnosing a problem invariably requires analyzing all the information collected from the cloud as well as from tools to manage the application lifecycle into a centralized storage. Typically, the troubleshooter has two choices: look at a variety of dashboards to try to understand the problem, or directly query the data. The vast amount of data collected from all distributed components of a cloud application, spanning several hosts and possibly different data centers, is overwhelming. The *runtime information* gathered includes: log statements; snapshots of cloud guests' state; system metrics, e.g., CPU utilization and memory consumption; and application metrics, e.g., response time and session length.

Since problems that occur in production often have their root in development-time decisions and bugs [Hamilton, 2007]—especially if new code is deployed frequently without appropriate test coverage—runtime data needs to be correlated to development-time changes for troubleshooting. However, actually doing so is



(a) Traditional problem diagnosis requires collecting fragmented information through a multitude of tools and data sources. A hint in one fragment leads to information in another, and so on.

(b) Context-based Analytics organizes the plethora of runtime information in form of a graph that relates relevant fragments to each other.

Figure 5.1: Contrasting traditional problem diagnosis with our approach of context-based analytics.

challenging for developers [Cito et al., 2015b], and involves inspecting multiple fragments of disperse information. An example is depicted in Figure 5.1a. After becoming aware of a problem via an alert, a developer investigates and manually correlates logs and metrics, until she finds that a specific change seen in the version control system is the likely culprit. The developer still needs to look at the change and start debugging the code. This procedure requires knowledge of tools and processes to obtain information, and perseverance to identify relevant fragments from heterogeneous data sets. Furthermore, when exploring the broader context of a problem, developers build mental models that incorporate source code and other collected information that are related to each other [LaToza and Myers, 2010]. The mental model consists of different information fragments, derived from the application, that are *implicitly* connected. Deployment, runtime and development knowledge are required to establish the links between the fragments within this mental model.

In this paper, we propose an analytics approach that incorporates system and domain knowledge of runtime information to establish *explicit links* between fragments to build a *context graph*. Each node in the graph corresponds to a fragment (e.g., individual logs, metrics, or source-code excerpts). Edges correspond to semantic relations that link a fragment to related fragments, which we refer to as its *context*. Figure 5.1b illustrates context-based analytics as compared to the traditional approach in Figure 5.1a. It organizes the plethora of runtime information in a context graph where developers navigate the graph to inspect relevant connected fragments. The nature of graph nodes and its relations are defined in an initial modeling phase during which a context model is built. The graph is then constructed on-line from the modeled data sources.

We implemented a proof-of-concept prototype of the context-based analytics framework. Furthermore, we instantiated our prototype with an application-context model for a real cloud application at IBM. We applied our framework to diagnose two issues taken from the issue tracker of the studied application and found that our approach reduced the effort of diagnosing these issues. In particular, it decreased the number of required analysis steps by 48% and the number of needed inspected traces by 40% on average as compared to a standard diagnosis approach.

This paper makes the following contributions: (1) we described an approach to unify runtime traces and source code in a graph structure and to explicitly establish connections between the information fragments, which we refer to as *context-based analytics*; (2) we implemented the approach, provided a reference architecture description and open sourced the framework on Github; and (3) we conducted a case study with a real cloud application.

Next, we contrast our approach with related work (§ 5.2), describe the framework for context-based analytics (§ 5.3), elaborate on the framework implementation (§ 5.4), delve into a case study with a real cloud application (§ 5.5), discuss the lessons we learned (§ 5.6) and limitations of our study (§ 5.7), and present some final remarks (§ 5.8).

5.2 Related Work

Our work falls within the general topic of software analytics, and is particularly related to research on traceability and visualization of runtime behavior.

5.2.1 Software Analytics

There is a multitude of work that can be combined as pertaining to software analytics [Menzies and Zimmermann, 2013]. While most research in this area has investigated how to use static information (e.g., bug trackers and code repositories) to guide decisions, some studies rely on runtime traces to provide insights for software engineers. Often log analysis is used to understand system behavior [Fu et al., 2013, Lin et al., 2016]. A recent study from Microsoft investigates how event and telemetry data is used by various roles in the organization [Barik et al., 2016]. These works usually focus on identifying one specific aspect of failing systems (e.g., performance [Yuan et al., 2014], emergent issues [Lin et al., 2016]), whereas our approach constructs a graph for systematically exploring runtime information.

5.2.2 Traceability

There has been extensive research on traceability between requirements (and other textual artifacts) and source code. Marcus and Maletic establish traceability links between documentation and source code using latent semantic indexing [Marcus and Maletic, 2003]. Spanoudakis et al. use a rule-based approach to infer these links [Spanoudakis et al., 2004]. In contrast, our work focuses on tracing various kinds of runtime information (including textual artifacts, such as log statements) to source code. The difference is mostly that runtime information (e.g., log statements) are much shorter than comparable software documentation and do not require such rigorous pre-processing as longer requirements documents. Linking runtime artifacts with code is probably more related to tracing in a performance modeling context [Heinrich, 2016].

5.2.3 Visualization of Runtime Behavior

Both systems and software engineering research have looked into different ways to understand runtime behavior through visualization. Sandoval et al. [Sandoval Alcocer et al., 2013] investigate performance evolution blueprints to understand the impact of software evolution on performance. Bezemer et al. [Bezemer et al., 2015] investigate differential flame graphs to understand performance regressions. Cornelissen et al. [Cornelissen et al., 2011] showed that trace visualizations in the IDE can significantly improve program comprehension. ExplorViz [Fittkau et al., 2015] provides live trace visualization in large software architectures. While existing work mostly focuses on a specific area of runtime information and one data source (e.g., performance as execution time from profilers or internal runtime behavior of objects from the JVM), we provide a framework to unify a diverse set of data from different data sources as a way to guide navigation on this search space.

5.2.4 State of Practice.

Industry tooling related to our approach mostly combines different metrics in dashboards. Probably the most prominent open-source tool chain in this area is the ELK stack[1] (ElasticSearch, Logstash, Kibana) where logs from distributed services are collected by Logstash, stored on ElasticSearch, and visualized in Kibana. More closely related to our approach is the open-source dashboard Grafana[2], that is mostly used to display time series for infrastructure and application metrics. Commercial counterparts to these services include, for instance, Splunk[3], Loggly[4], and DataDog[5]. The critique to the common dashboard solutions in current practice is that the amount of different, seemingly unrelated, graphs is overwhelming and it is hard to come to actionable insights [Cito et al., 2015b]. Our approach attempts to give guidance to navigate the plethora of

[1] https://www.elastic.co/webinars/introduction-elk-stack
[2] http://grafana.org/
[3] https://www.splunk.com/
[4] https://www.loggly.com/
[5] https://www.datadoghq.com

data by establishing explicit links between them. Further, to the best of our knowledge, DataDog is the only tool that attempts to correlate commits to other performance metrics. However, this correlation is based on temporal constraints of the commit only (similar to an idea in our own earlier work [Cito et al., 2016]) and does not involve any analysis of the code itself.

5.3 Conceptual Framework for Context-based Analytics

Figure 5.2 provides an overview of the components and process of context-based analytics. Our approach makes previously implicit links between between runtime information and code fragments explicit by constructing a graph (*context graph*) based on an application context model. The application model is an instantiation of a meta-model, that defines the underlying abstract entities and their potential links of context graphs. The modeling process is only required as an initial step. With every problem diagnosis task, a graph is constructed with the current runtime state of an application. Selecting a node leads to its context being expanded that yields more nodes that are linked to the initial one. Finally, each node is visualized based on its feature type.

In the following, we describe the components of the framework in more detail.

5.3.1 Meta-Model

We define a meta-model that defines abstract types and their possible relationships in the context graph that is depicted in Figure 5.2.

Runtime Entity is an abstract structure that includes any information that represents the state of an application and its underlying systems at a certain point in time $t = \{1, \ldots, T\}$. State is gathered either through observation (e.g., active monitoring [Cito et al., 2014a] or system agents) or through log statements issued by the application. A runtime entity consists of a number of attributes $e_t = \{a_1, \ldots, a_n\}$ that represent the application state at time t. In our meta-model, we differentiate between unit-, set-, and time series entities:

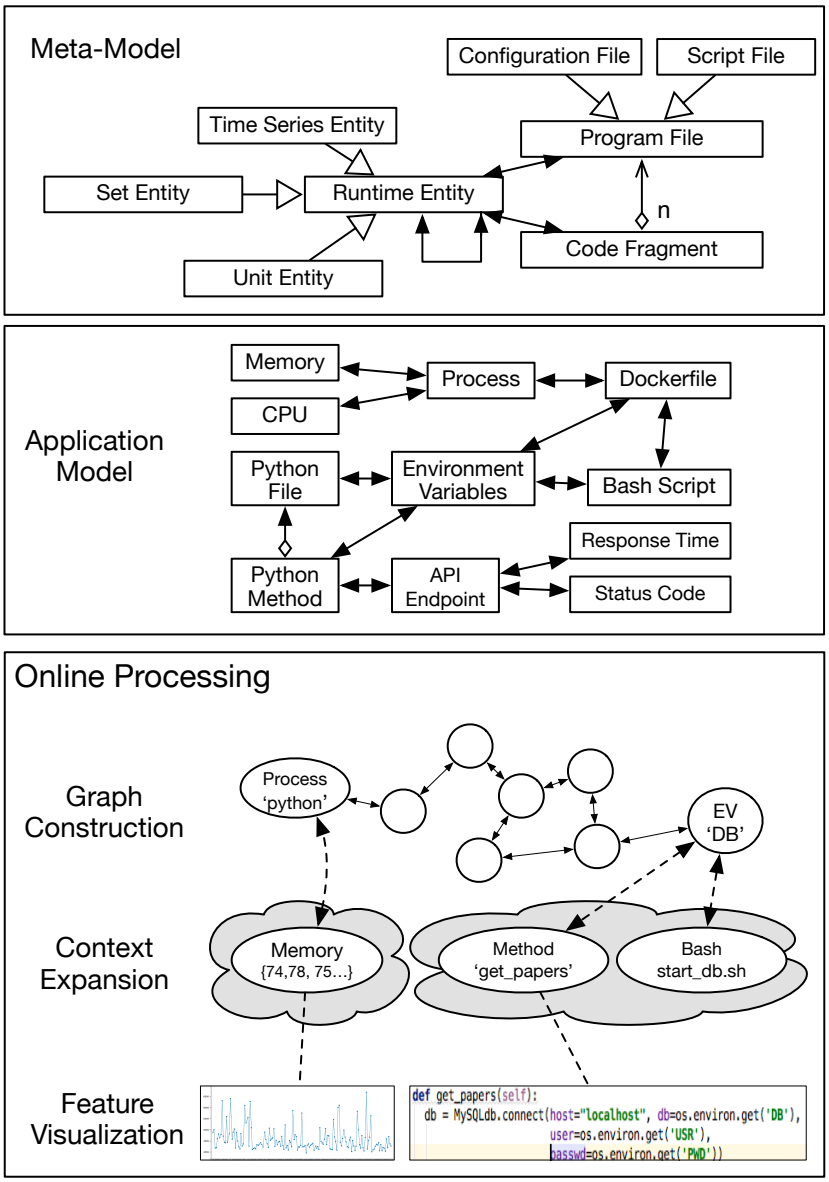


Figure 5.2: Overview of the context-based analytics approach from initial modeling to online processing of a context graph

- *Unit Entity* refers to a single fact that is observed at time point t that does not belong to a group of observations (such as a set or a time series).

- *Set Entity* refers to a set of unordered observations at a time point t that exhibit common properties. An example of a set entity could be a set of processes running within a container at a certain point in time.

- *Time Series Entity* refers to a set of totally ordered observations starting at time t within the time window w. An example of a time series entity could be the evolution of execution times of a REST API endpoint within a time window.

A runtime entity can establish links to other kinds of runtime entities and *Program Files* and *Code Fragments*.

Program File represents an abstract entity that encompasses all sorts of code files that relate to the application and are stored in version control (e.g., a Java file containing a class definition). A *code fragment* is a continuous set of code lines that is a subset of a program file.

We further differentiate between *configuration file* and *script file*. This distinction is modeled to enhance the basic capabilities of automatically establishing links.

- *Configuration File* refers to program files that set (initial) parameters for the application. This can also include infrastructure setup files (e.g., Dockerfiles). For a program file to qualify as a configuration file, it has to exhibit clear syntax and semantics towards providing configuration parameters (e.g., key-value pairs in .ini files, exposing ports in Dockerfiles).

- *Script File* refers to files that cannot be distinguished as either program files that yield application functionality, configuration, or operations task (e.g., batch processing).

Establishing Links

Two entities are linked through a mapping $\mathcal{L} : \mathcal{E} \times \mathcal{E} \mapsto [0; 1]$, that represents the degree of connectedness between features as a normalized scale between 0 (not connected at all) and 1 (very high level of connectedness). Based on the abstract entities defined in the meta-model, our approach encompasses basic

implementations of the function $\mathcal{L} : \mathcal{E} \times \mathcal{E} \mapsto [0; 1]$ to establish a link between two entities in $\mathcal{E}$[6]. On a high level, we follow the notion of semantic similarity in taxonomies [Resnik, 1995] to describe relationships between entities. The more information two entities have in common, the more similar they are. For entities $e_i \in \mathcal{E}$ with attributes $a_1, \ldots a_n$, this can be expressed as

$$\mathcal{L}(e_1, e_2) = \max_{a_i, a_j \in S(e_1, e_2)} sim(a_i, a_j) \tag{5.1}$$

where $S(e_1, e_2)$ is a set of attributes contained in both e_1 and e_2 where a similarity function, sim, for their respective attribute type exists. In the following, we describe basic implementations to establish links.

Time Series to Time Series Observations of metrics over time exhibit system behavior. Often a spike in a high-level metric is caused by one or several lower level components. This phenomenon is investigated by the means of correlating time series. Our approach incorporates basic time series correlation methods that can be parameterized in its correlation coefficient.

Example: Both CPU utilization and method response time are time series features of the system. Time series correlation analysis is applied to establish whether there might have been influence of CPU on response time. Note that, this does not necessarily establish a causal relationship.

Set/Unit to Code Fragment In addition to establishing links between runtime traces, we also want to trace runtime information back to source code. To perform this matching, we distinguish between code fragments as an AST representation and code fragments as plain text:

- AST Representation: If we can parse the code fragments, we perform matching on the AST representation of the source code. For each node

[6]We use the term “entities" to describe both runtime entities and program files and code fragments

that contains text (variable names, method calls, strings literals, etc.) in the AST, we compare to attribute values of the set items or unit through string similarity.

- Plain Text Representation: In other cases, attribute values of the set items or unit are compared to the whole program fragment text through string similarity.

Example: An environment variable is used to configure aspects of a system. To properly assess the ramifications of their respective values in an application, a developer has to inspect the code in which the environment variable is used. We can link these features by matching the variable name in an AST node of a method or script.

For the remaining combinations of entity types, we attempt to map attributes of two features by attribute name. If entities $e_1, e_2 \in \mathcal{E}$ contain attributes with the same name, we apply string similarity on the attribute values of those who matched in name.

5.3.2 Application Model

The application model is an instantiation of the meta-model. It describes all information required to construct the context graph for problem diagnosis. More specifically, the application needs to describe

- Concrete Runtime Entities with a unique name and a set of attributes, and its type derived from the meta model
- Query to a data provider (e.g., SQL query, API call)
- Optionally, a method to extract attributes from an unstructured query result (i.e., feature extraction [Jiang et al., 2008])
- Source Code Repository to extract program files
- Specification of links between entities

- If necessary, similarity measures for specific entity relationships

Given this information in the model, we can construct the context graph.

5.3.3 Context Graph

A context graph is a graph $G_t = \langle \mathcal{V}_t, E_t \rangle$ where nodes V_t describe the entities and edges E_t describe links. An edge between two nodes exists if the mapping $\mathcal{L}$ between the features yields a connectedness level above a certain threshold τ. All nodes connected through outgoing edges are called the *context* of the node.

Graph Construction

The aim of a context-graph is to support system comprehension and problem diagnosis by providing a representation of the problem space around application state and code fragments that can be *explored*. Due to the size of the problem space, manually inspecting and constructing the whole graph in a reasonable time is practically infeasible. Instead we opted for initially constructing the graph with a subset of "starting nodes", $V_S \in V$, that represent an entry point to the system. The starting nodes can be selected during the initial context modeling phase (as a default) or can be re-configured before the start of a diagnosis session. In the construction phase, the starting node values are retrieved from the data sources specified in the application model. Program fragment nodes are extracted from the code present in version control at the specified time t.

Example: In Figure 5.2, we see an example of two starting nodes: The process 'python' and the environment variable 'DB'.

Context Expansion

To continue with the exploration of the problem space in the graph, we rely on a function that given an entity node $e_i \in \mathcal{E}$ in the context-graph can infer a set of context nodes

$$\Gamma(e_i) = \{\ e_j \mid \mathcal{L}(e_i, e_j) > \tau\ \} \tag{5.2}$$

where τ is a pre-defined threshold to specify a minimum level of similarity for the features to qualify as connected.

The new nodes form a subgraph, the *context* of e_i,

$$C(e_i) = \langle V(e_i), E(e_i) \rangle \tag{5.3}$$

of the initial graph G_t^w with

$$V(e_i) = \Gamma(e_i) \tag{5.4}$$

as nodes and

$$E(e_i) = \{ \ (e_i, e_j, v) \mid v = \mathcal{L}(e_i, e_j), v > \tau \ \} \tag{5.5}$$

as edges.

Example: Referring back to Figure 5.2, we see an illustration how context expansion works. By selecting the node *Process 'python'* the memory consumption in the service that is related to that process is linked. In a similar manner, when selecting the node for *Environment Variable 'DB'*, context expansion establishes a link with the code fragment that contains the method 'get_papers' and the Bash file 'start_db.sh'.

Entity Visualization

The framework provides standard visualization for every component of the meta-model to support analysis. However, if an entity requires a specific visualization, the framework provides extension points to override the standard visualization.

5.4 Implementation

We implemented a proof-of-concept of the context-based analytics framework with a backend in Python. The frontend works as a combination of the Jinja2[7] template engine and JavaScript with the d3 visualization library[8] and Cytoscape.js[9] to display and manipulate the context-graph. Figure 5.3 shows a screenshot of the implementation.

Basic Abstractions. The framework provides the abstract base entities described in the meta-model in Figure 5.2. It also provides abstractions for data queries to support modeling and inference of the context-graph. Currently, the framework provides data adapters for Elasticsearch and JSON (retrieved from the file system or over HTTP) for application state. It can retrieve code fragments over Git and attempts to provide an AST for code fragments with ANTLR [Parr, 2013]. So far, the implementation has only been tested with Python code of the case study application. However, it should work for any available grammar for ANTLR.

Extension Points. The implementation is architected to support extensibility. It provides the basic structures described in the conceptual framework in Section 5.3 with extension points to either extend or override the functionality for (1) entity-to-entity similarity measures, and (2) visualizations. For instance, given a very specific entity that is unique to an application (see “Adaptation" entity in RQ2 of Section 5.5).

Components of the framework implementation are open-source and can be found online on GitHub[10].

5.5 Case Study

The goal of our case study was to evaluate to what extent the concept of context-based analytics can be used to diagnose runtime issues in real-life applications.

[7] http://jinja.pocoo.org/
[8] https://d3js.org/
[9] http://js.cytoscape.org/
[10] https://github.com/sealuzh/ContextBasedAnalytics

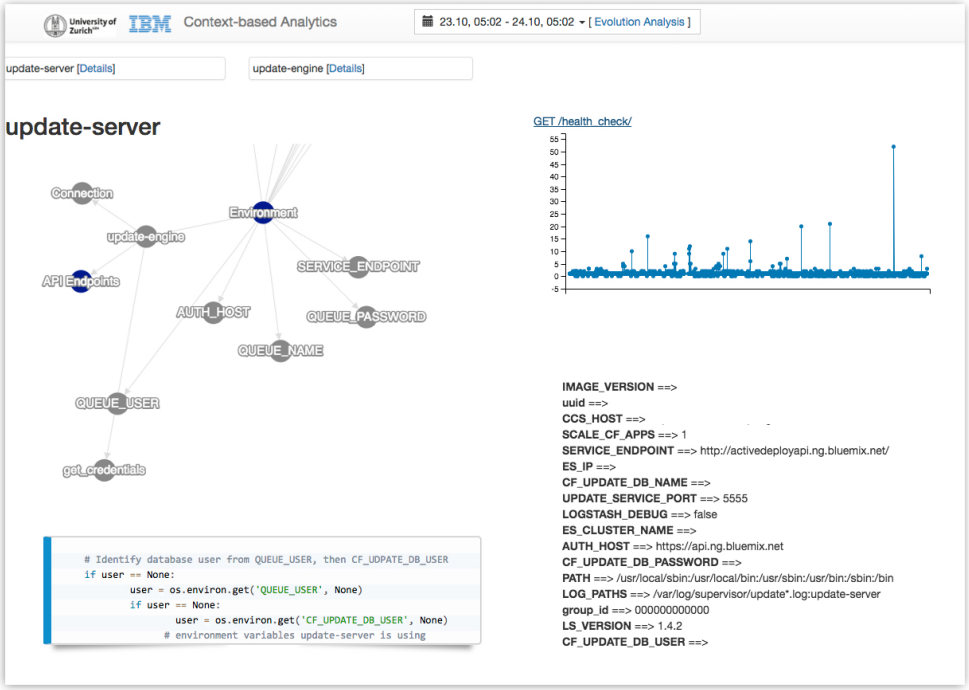


Figure 5.3: Screenshot of the context-based analytics tool implementation. On the left the context graph and around it visualizations of a time series entity, set entity, and a code fragment. On the top right developers can set time t and window w

Concretely, we investigate two research questions based on the proof-of-concept implementation discussed in Section 5.4.

1. RQ1 (Effort): How much effort (as measured in diagnosis steps taken and traces inspected) is required to diagnose a problem using context-based analytics as compared to a standard approach?

2. RQ2 (Generalizability): Can the context-based analytics approach also be applied to other case studies?

We have evaluated RQ1 based on IBM's *active-deploy*[11] application, which is part of the IBM Bluemix Platform-as-a-Service (PaaS) environment. We have

[11]https://github.com/IBM-Bluemix/active-deploy

chosen two real-life issues that have occurred within *active-deploy* to measure diagnosis efforts. To address RQ2, we have conducted the *application modeling* step (see Figure 5.2) for a second case study application from the CloudWave European project [Bruneo et al., 2014].

5.5.1 RQ1 – Evaluation of Diagnosis Effort

Case Study Application

The case study application used to address RQ1 is IBM's *active-deploy*. *active-deploy* is a utility application that allows the release of a new cloud application version with no down time. It is implemented as a service-based architecture and consists of approximately 84 KLOC. Each individual service runs in its own Docker container. Runtime information is gathered in the form of system metrics and log messages. System metrics are observed through an agentless system crawler[12]. The metrics are available in a homegrown metric storage service over an API. Logs from the distributed services are collected through a log aggregator (Logstash) and stored on a central database (ElasticSearch).

Case Study Application Modeling

The framework implementation (see Section 5.4) provides basic components for (1) abstract entities in the meta-model, (2) interfaces to common data providers (e.g., JSON over an HTTP API), and (3) common visualizations for the entities in the meta-model (e.g., line chart for time series, syntax highlighting for code fragments). The application model is an instantiation of these basic components of the meta-model and defines entities with data sources and their links. For *active-deploy*, we depict the entities and the links of the application model in Figure 5.2. Every entity models a query through a data provider (e.g., SQL query, API call). We wrote a data provider for ElasticSearch and the IBM internal metric storage service. Further, we needed to provide the model with a `git` repository to extract code fragments.

[12]https://developer.ibm.com/open/agentless-system-crawler/

In the following, we describe how we used this model to compare the effort of context-based analytics to a traditional baseline setting, in real-life problem scenarios.

Problem Scenarios

To study our approach based on realistic situations, we must evaluate them for runtime issues that are representative of common real-life situations. We are aware of two seminal works that discuss the underlying reasons of runtime problems in Web- and cloud-based software. Firstly, Pretet and Narasimhan conclude that 80% of the failures are due to software failures and human errors [Pertet and Narasimhan, 2005]. Secondly, Hamilton has postulated that 80% of all runtime failures either originate in development, or should be fixed there [Hamilton, 2007]. Hence, we have sampled the issue tracker of *active-deploy* and have selected two example runtime problems (scenarios) with roots in software development that are considered "typical" by application experts from IBM. We have excluded problems that are purely operational, e.g., issues caused by hardware faults. In compliance with requirements from IBM with regards to not revealing the internal technical architecture of the case study application, we refer to the two relevant services within the case study application as *Service* 1 and *Service* 2.

Scenario 1 - Context: Environment Variable Configuration Mismatch. Service 1 and Service 2 relay messages to each other through a document database that acts as a message queue. Service 1 acts as the frontend to the application and writes tasks into the queue that Service 2 reads from. Service 2 (which is scaled horizontally to multiple cloud instances) writes heartbeat messages into the queue. Service 1 reads the heartbeat messages to determine the availability of Service 2 as the task executor. The engineers in the team receive an alert from operations indicating that *active-deploy* is down and need to investigate. The root cause turned out to be a mismatch of an environment variable in the code of Service 2, which led to the creation of a document database with the mistyped name.

Scenario 2 - Context: Service Performance Regression. Service 1 provides a REST API to allow cloud applications to be upgraded with zero

downtime. The engineers in the team receive a call that interactions with cloud applications are significantly slower than usual. From the information available to the team it is not clear which specific REST API endpoint is affected. There have been no recent changes to any of the endpoint's source code. After investigation, the root cause has been narrowed down to an additional expensive call added previously, which has only now started affecting the service performance due to changes in the service workload (higher usage).

We further divide those scenarios into 8 concrete subtasks (i.e., questions that an engineer diagnosing these problems needs to answer), which we have designed in collaboration with application experts. These subtasks are listed in Table 5.1.

	Environment Variable Configuration Mismatch
T1	How long has the application been down?
T2	When did Service 1 last communicate to the queue?
T3	Is Service 2 alive and sends heartbeat messages to the database?
T4	Is Service 2 processing tasks from the database?
T5	How and where are the environment variables for the database set?
	Service Performance Regression
T6	When did the performance regression start?
T7	What endpoints are affected?
T8	What code has been changed on that endpoint?

Table 5.1: Subtasks used in the diagnosis of two real runtime issues in *active-deploy*.

Methodology

In RQ1, we aim to establish the effort that an engineer has to go through to diagnose the two scenarios, including all 8 subtasks. We first design a company baseline case study as described by Kitchenham [Kitchenham, 1996]. This baseline includes all data sources, tools, and workflows the IBM team who developed the case study application used for problem diagnosis. We then compare our

context-based analytics approach to this baseline using two metrics:

# of steps taken counts all distinguishable activities taken that might yield necessary information to be able to reach a conclusion of the given task. In the basline, steps include, but are not necessarily limited to:

- *Data Query*: Issuing a query to a data source (either within a log dashboard or directly in the database). This activity also includes refining and filtering data from a previous data query.
- *Data Visualization*: Plotting or visualizing data points.
- *Inspecting File Contents*: Opening a file to inspect it. This can be a file that is either in version control or on an operational system (e.g., log files).
- *Software History Analysis*: Inspecting the software history in version control (i.e., commit details) including their changes (e.g., changed files and diff's) [Codoban et al., 2015].

Contrary, in context-based analytics, steps taken translates to expanding the context on a node in the graph (which in the background translates to many steps, that would have been taken manually in the baseline), or adjusting the time t or window w of observation.

# of traces inspected counts all information entities that have to be investigated to either solve the task or provide information that guide the next *steps to be taken*. Information entities in this context include, but are not limited to:

- *Log statements*: A log statement either in a file or within an aggregated dashboard (e.g., Kibana).
- *Graphical Representation of Data Points*: A plot/visualization of a time series, histogram, or similar, where the inspection of the graphical representation as a whole (as opposed to inspecting every single data point in the graph) leads to observing the required information.

- *Datastore Entries*: A row resulting from a query to a data store (including aggregation results).

The first author of the paper performed all subtasks for both, the company baseline and the context-based analytics tooling, and manually tracked the steps taken and traces inspected. The evaluation procedure has been validated by application experts. We provide a log of all actions taken during the case study and its detailed description in an online appendix[13]. For the baseline, all subtasks have been performed so that the number of steps taken and traces inspected are minimal based on the information provided by application experts. Hence, the following results represent a conservative lower bound of the benefits of context-based analytics.

Results

The results of this case study evaluation are summarized in Table 6.3. Over all 8 tasks combined, our approach saved 48% of steps that needed to taken and 40% of traces that needed to be inspected. There is only a single subtask (T4) where using context-based analytics leads to a (slightly) increased number of traces that need to be inspected. For 3 subtasks (T3, T6, and T7), the number of inspected traces is unchanged. However, the number of steps taken increases substantially for all subtasks.

Note that these numbers represent an idealized situation and do not contain any unnecessary diagnosis steps or traces for either the baseline or our approach (i.e., the evaluation assumes that developers never run into a dead end while diagnosing the issue). Further be aware that a "step" in both approaches constitutes different things. In the baseline setting, a step can be a complex query to a database, fine-tuned to retrieve specific results. Conversely, in our approach, a "step" is only expanding a context node in the prototype implementation, or setting a different time t or window for time series data w. Further, specific domain knowledge is often necessary to construct proper queries to available data sources. Through our approach, this domain knowledge is also required,

[13]Online Appendix: https://sealuzh.github.io/ContextBasedAnalytics/

	Baseline		**CBA**		**Effort Comparison**	
Task	Steps	Traces	Steps	Traces	Δ Steps	Δ Traces
T1	4	2	2	1	-2 (50%)	-1 (50%)
T2	2	3	1	2	-1 (50%)	-1 (33%)
T3	2	2	1	2	-1 (50%)	0 (0%)
T4	2	7	1	8	-1 (50%)	+1 (-12.5%)
T5	5	42	3	18	-2 (40%)	-24 (43%)
T6	5	2	2	2	-3 (60%)	0 (0%)
T7	2	18	1	18	-1 (50%)	0 (0%)
T8	3	17	2	5	-1 (33%)	-12 (70.5%)
Total	**25**	**93**	**13**	**56**	**-12 (48%)**	**-37 (40%)**

Table 5.2: Case study results for RQ1. Using context-based analytics, the number of diagnosis steps that need to be executed are reduced by 48%, and the number of traces that need to be inspected are reduced by 40%. These numbers represent improvements over an idealized baseline, where engineers do not "waste time" with unnecessary diagnosis steps or traces that are not relevant to the issue at hand.

but encoded in the initially constructed application context model. All in all, we expect the real-life effort savings of using context-based analytics to even be underrepresented by the above numbers, especially for a novice engineer, or an engineer who is not an expert of the application that should be diagnosed.

5.5.2 RQ2 – Evaluation of Generalizability

To get an idea whether the meta-model of context-based analytics can be generalized, we conduct the application modeling step for another industrial application.

Case Study Application

The case study is a larger application from a major telecommunications provider that is deployed within the CloudWave European project [Bruneo et al., 2014]. It was chosen as a representative instance of a cloud application with distributed logs tracking multiple scalable components. The project is implemented as a service-based architecture and consists of approximately 139 KLOC. It is deployed

in an OpenStack environment[14]. Runtime information are gathered in the form of system metrics, log messages, and application metrics. System metrics are mostly gathered through the underlying platform, OpenStack and stored in the service Ceilometer[15], that acts as a “metric hub". Application metrics are sent directly from the application through a low-level system daemon (similar to StatsD[16]) where they are stored in a document database (MongoDB). Further, application metrics are also derived from low-level metrics that are aggregated through complex-event-processing in Esper [Leitner et al., 2012]. These are then relayed through a homegrown tool called CeiloEsper[17].

Case Study Application Modeling

We modeled this additional case study by retrieving a list of data sources and its metrics as *metric name, metric type, metric unit*. We established the links between entities based on our understanding of the application and discussed the results with application experts and operators of the CloudWave project. The entities and links of the resulting application model are depicted in Figure 5.4.

Comparison

The model has some overlap with the model of our first case study at IBM, mostly regarding lower level machine metrics (e.g., CPU, Memory). Being an application in the telecommunications industry, it has an emphasis on infrastructure and network metrics both on systems level (the “Network" entities on the top right in Figure 5.4 include incoming and outgoing bytes/packets/datagrams, read and write rates of bytes, active connections, etc.) and application level (e.g., jitter, average bitrate, number of frames). *Adaptation*s are important events in the domain of the application [Bruneo et al., 2015] (application-specific functionality degradation). Diagnosing why specific adaptations have happened in CloudWave is well supported by context-based analytics, as adaptations are triggered by

[14]https://www.openstack.org/
[15]https://github.com/openstack/ceilometer
[16]https://github.com/etsy/statsd
[17]https://github.com/MDSLab/cloudwave-ceilometer-dispatcher

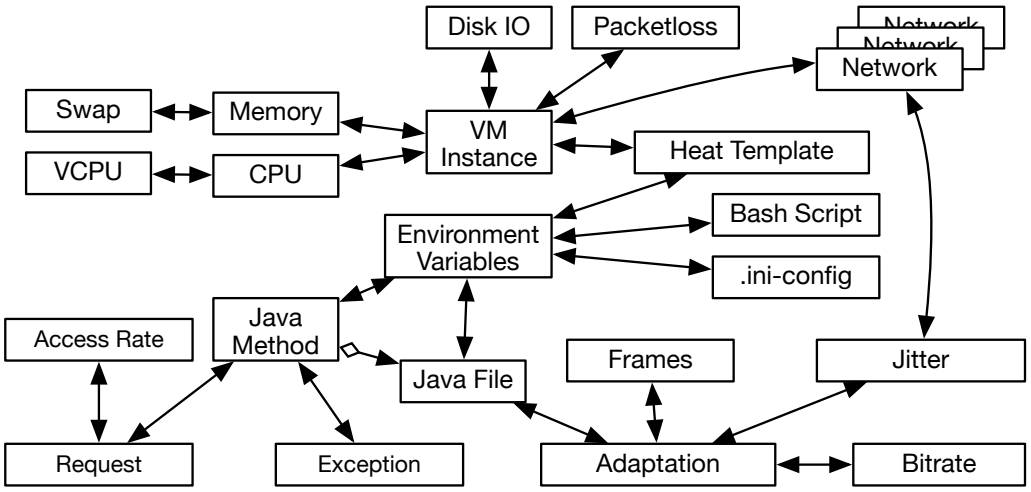


Figure 5.4: Application model of an application in the telecommunication industry deployed in OpenStack from the CloudWave European project

the application metrics *Jitter*, *Bitrate*, and *Frames*. We can also establish a link back to the Java code file, since the adaptation actuators are marked in the code through Java annotations.

The abstractions in the meta-model proved flexible enough to appropriately model both case study applications from vastly different domains. However, the visualizations provided by the current proof-of-concept implementation built for IBM are limited in their expressiveness, and not generally useful to the CloudWave implementation without changes. For instance, *Adaptation* in this case study is a time-series entity, but a line chart of simply a boolean value (adaptation happened or not) over time can be improved. However, the implementation counters this by offering extension points to assign different visualizations to specific entities.

5.6 Discussion

Based on our case study implementation and discussions with practitioners, we have identified a number of interesting challenges and discussion items in the scope of context-based analytics.

The need for dynamic context models. In the current state, the application context model is designed once as part of an initial modeling phase, and then instantiated throughout the use of our approach. This is especially favorable for junior engineers and newcomers to the project, as it allows them to systematically browse traces with little to none of the required domain knowledge. However, in the course of our case study, it became apparent that allowing dynamic queries to the existing data repositories [Sun et al., 2014] to extend the context model on the fly can be beneficial for more senior engineers. Otherwise, our approach faces the danger of being perceived as a rigid corset that limits experts rather than supporting them. Hence, as part of future work, we will investigate an extension to the context-based analytics meta-model to support the creation of a dynamic, ad-hoc context model.

Integrating business metrics. Our framework conceptually allows for the integration of various kinds of runtime data. However, so far, we have exclusively focused on the aggregation of technical runtime data (e.g., response times, CPU utilization) to static information, such as code changes. An interesting extension of this idea is to also integrate metrics that are more related to the business value of the application, for instance click streams, conversion rates, or the number of sign-ups. We expect that our meta-model is already suitable to allow for integrating business metrics. However, whether other extensions are practically necessary is an item for future investigations.

Data privacy. To construct the context graph we solely use information that is already available through existing channels in the organization. We only establish explicit links that beforehand were implicit, and, thus, probably harder to retrieve. However, automated event correlation and data aggregation can still be the cause of privacy concerns. In previous work, we argued for collaborative feedback filtering as part of organization-wide governance [Cito et al., 2015b].

Collaborative, privacy preserving data aggregation has been discussed as a solution to this problem in the past [Applebaum et al., 2010, Burkhart et al., 2010].

5.7 Threats to Validity

We now discuss the main threats to the validity of our case study results.

Construct Validity. We have constructed our evaluation regarding RQ1 and RQ2 carefully following the established framework introduced by Kitchenham [Kitchenham, 1996]. However, there are still two threats that readers need to take into account. Firstly, we have chosen *# of steps taken* and *# of traces inspected* as observed metrics in RQ1 to represent the effort that engineers have to go through when diagnosing runtime issues. These metrics are necessarily simplified to allow for objective comparison. However, in practice not all steps and traces are the same, e.g., some diagnosis steps may be more difficult and time-consuming than others. Secondly, there is the threat that we have overfitted our general context-based analytics framework or prototype implementation for the specific *active-deploy* use case. We have mitigated this threat by additionally applying our approach to a second use case as part of our evaluation of RQ2.

Internal Validity. For RQ1, the first author manually retrieved the observed metrics, which is subject to evaluator bias. We mitigated this threat by documenting the process in detail. Further, the second author, who is also an application expert of the case study application, has independenly validated the diagnosis steps of the baseline setting. Additionally, we are as explicit as possible in describing how the tasks are being evaluated, and provide the documented process as an online appendix.

External Validity. A fundamental question underlying all case study research is to what extend the results generalize to other comparable applications (e.g., multi-service cloud applications outside of IBM). This threat was the main reason for our investigation of RQ2. While we were unable to report on the second case study in the same level of detail as for the first case due to data confidentiality issues, we were able to develop a suitable application model following the context-

based analytics meta-model also for this second independent case, without requiring changes to the meta-model. Another threat to the external validity of our results is the question whether the two real-life issues selected in our evaluation of RQ1 are representative of common problems. We have mitigated this threat by carefully selecting real issues in collaboration with application experts, and based on suggestions from earlier literature.

5.8 Conclusion

In this paper, we have addressed the problem of diagnosing runtime problems in large-scale software. We have illustrated that problems that are observed in production, such as outages or performance regressions, often originate in development-time bugs. However, actually identifying the problematic changes in program or configuration code requires inspecting multiple fragments of disperse information that are implicitly connected. Currently, multiple different tools need to be used in conjunction (e.g., log aggregation tools, source code management tools, or runtime metric dashboards), each requiring different data sources and workflows.

We proposed context-based analytics as an approach to make these links explicit. Our approach consists of a meta-model, which needs to be instantiated in an initial modeling phase for a concrete application, and an online processing phase, in which the context model is supplied with concrete runtime data. We instantiate our context model for two real cloud computing applications, one from IBM's Bluemix PaaS system, and one from a major telecommunications provider. Using two concrete practical runtime problems from the Bluemix-based application's issue tracker, we show that using context-based analytics we are able to reduce the number of analysis steps required by 48% and the number of inspected runtime traces by 40% on average as compared to the standard diagnosis approach currently used by application experts. Our future work will include investigating the dynamic creation of context models to support ad-hoc querying, the integration of business metrics in addition to technical runtime metrics, and the investigation of privacy concerns in our approach.

6

Supporting Software Development with Runtime Performance Data

Jürgen Cito, *Philipp Leitner, Harald C. Gall*
in submission to an international journal
Contribution: User study design, data collection, data analysis, prototype implementation, paper writing

Abstract

Performance problems are hard to track and debug, especially when detected in production and originating from development. Software developers try to reproduce the performance problem locally and debug it in the source code. However, data combinatorics of configuration in production environments are

too different to what profiling and testing can simulate either locally or in staging environments. Especially in systems with scalable infrastructure, software developers need to inspect distributed logs to analyze the issue. We propose an integrated approach that models runtime performance data and matches it to specific, performance-related elements of source code in the IDE (methods, loops, and collections). Given the information on the source code level, we leverage inference models that provide live feedback of performance properties of newly written code to prevent performance problems from even reaching production. We describe a proof-of-concept tool as an Eclipse plugin and evaluated it through a controlled experiment with 20 professional software developers, in which they worked on software maintenance tasks using our approach and a representative baseline (Kibana). We found that, using our approach, developers are significantly faster in (1) detecting the performance problem, and (2) finding the root-cause of the problem. We also let both groups work on non-performance relevant tasks and found no significant differences in task time. We conclude that our approach helps detect, prevent and debug performance problems, while at the same time not adding a disproportionate distraction for maintenance tasks not related to performance.

6.1 Introduction

Performance problems in production cloud-based applications often originate in development [Hamilton, 2007]. Tracing these problems back to source code and debugging them requires software developers to inspect runtime information (e.g., logs, traces, metrics). Software operations are tasked with collecting this information by observing software deployed in production environments to capture interactions and behavior at runtime through monitoring and instrumentation. Consequently, a large body of academic [Aceto et al., 2013, van Hoorn et al., 2012] and industrial work in the form of application performance monitoring tools (APM), e.g., New Relic or Dynatrace, exist that collect and visualize runtime performance data. In these tools, the information required to reconstruct complex scenarios and solve performance problems, comes in the form of time series graphs

in dashboards or numbers in reports. These abstractions are largely designed to support the work of software operations, whose responsibilities require them to think in systems, rather than the underlying source code. This traditional view of operations analytics does not fit into the workflow of software developers, which involves producing high volumes of program code in an Integrated Development Environment (IDE). Consequently, software developers struggle to incorporate this runtime performance data into their development and debugging activities. Existing work has argued that diagnosing performance regressions in cloud applications is currently often based on personal belief and gut-feeling [Devanbu et al., 2016], rather than on the hard data that would in principle be available.

Developer Targeted Performance Analytics. We argue that the existing state-of-the-art does not present runtime performance data on the right level of abstraction for software developers; they require tight feedback loops between source code and production runtime. In this paper, we propose *Developer Targeted Performance Analytics (DTPA)*, an approach to contextualize runtime data for software developers and make them more aware of the operational footprint of their source code. The idea of DTPA is to match runtime performance data with specific performance-relevant elements of source code (methods, loops, collections) and visualize it in the IDE. This allows software developers to *anticipate runtime issues* before they occur, as well as leverage *data-driven decision-making* based on runtime data collected from software in production.

DTPA serves three goals: (1) contextualization, i.e., bringing production feedback into the context and workflow of software development tasks, (2) raising awareness for operational footprints, and (3) a-priori preventing problems via inference of the performance properties of code changes prior to deployment. Of particular importance is that DTPA models data that has emerged out of actual production workloads, in comparison to previous related research that primarily aimed to visualize profiling data [Beck et al., 2013].

Contribution

The paper makes the following contributions:

- **Conceptual Framework**: It introduces a conceptual framework that combines aspects of program analysis and statistical inference to provide early feedback of software performance problems. Within this concept, it introduces an algorithm that can map runtime properties to low-level source code artifacts, infer performance properties of new code and propagate this new information in linear time.

- **Implementation (PerformanceHat)**: It presents *PerformanceHat*, a proof-of-concept implementation of DTPA for Java programs in Eclipse. PerformanceHat provides an interface to integrate runtime metrics from various sources and predict the execution time of new method invocations and newly introduced loops.

- **User Study Evaluation**: It presents a controlled experiment with 20 professional software developers to evaluate DTPA's effectiveness. We show that developers using DTPA were significantly faster in detecting and finding the root-cause of performance problems. At the same time, when working on non-performance relevant tasks, software developers with DTPA did not perform significantly different, indicating that integrating additional data in source code views does not lead to unnecessary distractions.

6.2 Related Work

Work related to this research can be broadly categorized as follows: (1) software analytics, (2) visualization of runtime behavior, and (3) change impact analysis for performance.

Software Analytics. Researchers have extensively investigated methods and approaches to mine software repositories for a variety of reasons and stakeholders under the term "software analytics" [Buse and Zimmermann, 2010]. Often, these analytics approaches provide prediction models to support software managers to make decisions within their teams [Misirli et al., 2013, Buse and Zimmermann, 2012, Minku et al., 2016]. One of the focal points of software analytics is bug and defect prediction. However, a study by [Lewis et al., 2013]

found that after deploying bug prediction techniques, there was no identifiable change in developer behavior (i.e., not significantly more bugs were found than before). [Zhang et al., 2013] also investigated the impact of software analytics approaches. In contrast to our work, most of the work in software analytics mines static information (e.g., source code repositories, issue trackers, mailing lists) to build prediction models. We focus on analytics based on runtime data, specifically software performance.

Visualization of Runtime Behavior. Work, both from software and systems engineering research, has investigated different ways to understand runtime behavior through visualization. [Sandoval Alcocer et al., 2013] present an approach called 'performance evolution blueprint' to understand the impact of software evolution on performance. [Meyer et al., 2016] visualize the process runs of stored procedures in database systems. Senseo [Röthlisberger et al., 2009] is an approach embedded in the IDE that augments the static code view perspective with dynamic metrics of objects in Java. [Bezemer et al., 2015] investigated differential flame graphs to understand software performance regressions. [Cornelissen et al., 2011] showed that trace visualization in the IDE can significantly improve program comprehension. ExplorViz [Fittkau et al., 2015] provides live trace visualization in large software architectures. Similarly to our work, [Beck et al., 2013] provide augmented code views with information retrieved from profilers. Our work differs first in the use of data retrieved through instrumentation in production systems. Further, our approach goes beyond displaying information on existing traces by providing live impact analysis on performance of newly written code.

Impact Analysis & Performance Prediction. Change impact analysis supports the comprehension, evaluation, and implementation of changes in software [Lehnert, 2011b, Bohner and Arnold, 1996]. Most of the work that is related to change impact analysis and performance prediction operates on an architectural level [Heger and Heinrich, 2014, Becker et al., 2009] and is not supposed to be “triggered" during software development. Recent work by [Luo et al., 2016] uses a genetic algorithm to investigate a large input search space that might lead to performance regressions. Our approach for change impact analysis

is applied live, i.e. during software development, and leverages an analytical model consisting of the immediate software change and the existing runtime information to provide early feedback to software developers.

6.3 Context

To specify the context and scope of our work, we present an illustrative example followed by some assumptions. The example is simplified for clarity, but inspired by discussions with our industrial partners and collaborators.

Motivating Example We consider a software developer who mostly works in Java within the Eclipse IDE. Her main project is the company's team-centric VoIP chat client (similar to Slack), where users log in and interact with multiple teams over chat and VoIP. This application is based on a service architecture, where each service is deployed as a scalable unit in the cloud. The deployed software is monitored to observe operational features and give insight into online application behavior.

One of the services of the application is the "LoginFrontend" service that provides authentication functionality. Users of the service are usually members of multiple teams. However, the current version only allows one team to log in at a time. A new change request asks to extend the functionality of the login for all teams of a user all at once. The software developer locates the appropriate method in the code (see Listing 1) and starts implementing the change request. She sketches out the naïve solution represented in Listing 2. It iterates through all users present in the UI elements (method `getUsers()`) and performs the login routine (method `new DataManger.start(username, password)`) in the loop body for each team of the user. She first tests the change manually with provided test data and runs unit tests for "Login Frontend". The result is satisfying and the change is pushed to Continuous Integration (CI), where it passes all quality gates (ranging from code review, automated tests, as well as synthetic load testing).

```
private void login() {
 VTextBox b = singleUserVBox;
 String username =
  b.user.getSelectedItem().toString();
 String pwd = b.getPassword();
 new DataManager(b).start(userName, pwd);
```

Listing 1: Original code of "login" method.

```
private void login() {
 for(String user : this.getUsers()) {
  VTextBox b = this.getVboxes().get(user);
  String username =
   b.user.getSelectedItem().toString();
  String pwd = b.getPassword();
  new DataManager(b).start(userName, pwd);
 }
```

Listing 2: New version with loop.

Unfortunately, after rollout to production, the developer receives an unexpected alert for the "LoginFrontend" service. The APM system reports the response time metrics for the login with > 20 seconds. Support also already receives complaints from customers about timeouts and slow logins.

Diagnosis. Even with many traditional testing in place (both functional and non-functional), the developer was not able to observe any non-linear responses and reproduce the production issue. Only after careful inspection of distributed operational logs in the company's APM system, she is able to find the culprit: the process of logging into a team also leads to eagerly loading the profile pictures of each individual team member from a 3rd party service (e.g., Facebook, LinkedIn). The 3rd party service was mocked for testing environments, further obfuscating the issue.

Developer-Targeted Performance Analytics. With our approach, DTPA, the operational footprint of the *DataManager.start* method invocation would have been contextualized in the software development workflow (i.e., the execution

time observed in production would have been visually attached to the method invocation). Additionally, inference of newly written code would have provided live feedback to the software developer, warning about a potential performance degradation.

6.3.1 Scope

Given the vast space of different software being built, we need to narrow down the application model and the assumptions underlying our work. In general, we distinguish between *conceptual* assumptions which are fundamental to the idea of DTPA, and *prototype* assumptions, which relate to how we have implemented our approach as described in Section 6.5. Consequently, prototype assumptions can potentially be relaxed or lifted in the future via different implementation decisions. A listing and explanation of these assumptions and context factors is given in the following.

- *Online Services* (Concept): We target online services that are delivered as a service (SaaS applications), mostly deployed on cloud infrastructure, which are accessed by customers as web applications over the Internet. Specifically, we do not consider embedded systems, "big data" applications, scientific computing applications, or desktop applications. While an approach similar to DTPA could also be used for a subset of those, the concrete modeling and performance-related challenges would change, making them out of scope for the present study.

- *Software Maintenance* (Concept): We assume that the application is already deployed to production and used by real customers. However, given the ongoing industrial push to lean development methodologies and continuous deployment [Savor et al., 2016], many SaaS applications are "in maintenance mode" for the vast majority of their lifetime.

- *Performance on System-Level* (Concept): DTPA is intended to support performance engineering on a systems level rather than improving, e.g., the algorithmic complexity of a component. Hence, a focus is put on component-

to-component interactions (e.g., remote service invocations, database queries), as these tend to be important factors contributing to performance regressions on a system level, while also being hard to test without knowing production conditions.

- *Production Observable Units* (Concept): Our approach is limited to modeling methods that are actively measurable by existing APM tools. Thus, methods that might have suboptimal (theoretical) computational complexity, but do not exhibit any significant overhead that is captured by monitoring will not be modeled.
- *OOP* (Prototype): The current prototype implements the components of the conceptual framework and provides proper extension points. Due to implementation details, the prototype currently only works with object-oriented programming languages.
- *Strong Type System* (Prototype): While in theory the mapping between source code artifacts and runtime traces can be established in many ways, our current implementation makes heavy use of type information on AST level to create the mapping, and hence builds upon a strong type system.

6.4 Concept

We propose a conceptual framework to model runtime data along with source code artifacts to achieve a tight integration of runtime aspects in the software development workflow. While writing code in the IDE, the program is transformed into a simplified Abstract Syntax Tree (AST), and, through a process called *feedback mapping*, runtime traces are matched with the appropriate source code nodes. The result is an AST annotated with runtime metrics that is visualized in the IDE. Additionally, live impact analysis for certain code changes provides early feedback on potentially performance degrading code. Our approach enables data-driven decision making as it provides performance awareness of the operational footprint based on collected runtime data in the context of a specific task within the program.

In the following, we provide more details to the underlying concepts of *feedback mapping* and *impact analysis*, after we introduce some preliminaries.

6.4.1 Preliminaries

We consider programs p as syntactically valid programs of a language $\mathbb{P}$. A program $p \in \mathbb{P}$ consists of a set of methods, $m \in M(p)$, where every method m is uniquely identifiable through $id(m)$ (e.g., fully qualified method names in Java) organized in classes (or any other unit of organization for methods, e.g., modules).

Abstract Syntax Tree. A syntactically valid program, $p \in P$, can be transformed into an Abstract Syntax Tree, a tree representation of the source code of p, denoted by $AST(p)$. We consider a simplified AST model where the smallest entity nodes are method declarations and statements within a method (method invocations, branching statements, and loops). Formally, an AST is a tuple (A, a_0, ζ), where A is set of nodes in the AST (*source code artifacts*), $a_0 \in A$ is the root node and $\zeta : A \mapsto A^*$ is a total function that, given a node, maps it to a list of its child nodes. Each node $a_i \in A$ has a unique identifier, $id(a_i)$. For convenience, we also define a function $AST_M(m)$ that returns the AST for a method $m \in M(p)$. Formally, $m_{AST} \subseteq p_{AST}$, where $m_{AST} = AST_M(m), p_{AST} = AST(p)$, $m \in M(p)$ and $id(m) = id(a_0)$ in m_{AST}.

6.4.2 Trace Data Model

Our approach relies on execution traces that have been collected at runtime, either through observation, instrumentation, or measurement. While this data could potentially have different types, the focus of this paper is on runtime data that is relevant to software performance (e.g., execution times, load). Let us consider a dataset $\mathcal{D}$ to be the set of trace data points. The model of a data point $d_i \in \mathcal{D}$ is illustrated in Table 6.1. The illustrative trace shown in the table is an actual trace type used in the evaluation through the monitoring tool Kieker [van Hoorn et al., 2012]. We model the point in time the data point has been measured or observed and the runtime entity that “produced" the data

point (the granularity ranges from methods to API endpoints to operating system processes). We assume every trace has a numerical or categorical value of the observation. Many traces are also associated with some kind of label that is part of the trace meta data (e.g., method name). The context is a set of additional information that signifies, for instance, on which infrastructure the entity was deployed on, or, which log correlation id (in the example it is called “SessionId") was involved [Jiang et al., 2009].

Table 6.1: Trace Data Model

Abstract	Description	Illustrative Trace
t	Recorded Time	Logging Timestamp
$\mathcal{E}$	Observed Entity	Java method
$\mathcal{T}$	Trace Type	Execution Time (ET)
$\mathcal{V}$	Primitive Value	Measured time (e.g. 250ms)
L	Label	Method Name (e.g., us.ibm.Map.put)
$\mathcal{C}$	Context	{Stack Size, SessionId, Host,...}

6.4.3 Feedback Mapping

In an initial step, AST nodes are combined with the dynamic view of runtime traces in a process called feedback mapping. This mapping constitutes a relation between nodes in the AST and a set of trace data. On a high level, this process is inspired by previous work in software traceability using formal concept analysis [Poshyvanyk and Marcus, 2007, Eisenbarth et al., 2003], which is a general framework to reason about binary relationships. In the IDE, this results in a performance augmented source code view, that allows developers to examine their code artifacts annotated with performance data from runtime traces from production environments (e.g., method calls with execution times, usage statistics for features, or collections with size distribution).

Figure 6.1 illustrates the process with source code from the motivating example (see Section 6.3). The code is first transformed to a simplified AST,

where each node is traversed and, if applicable, is assigned a set of trace data points. A set of traces can be mapped to different AST node types (e.g., method invocations, loop headers) based on different specification criteria in both node and trace. In this particular case of response times, we map data based on the fully-qualified method name in Java, that is both available as part of the AST node and in the trace data. Specifications define declarative queries about program behavior that establish this mapping relationship.

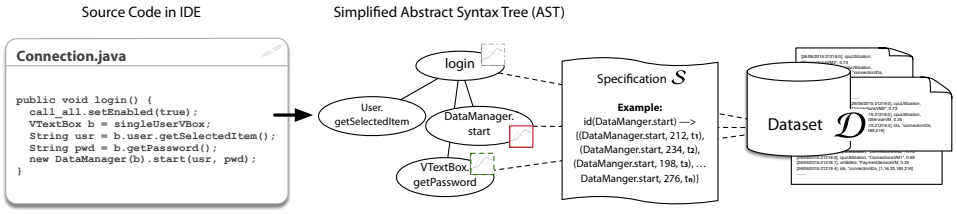


Figure 6.1: An illustration of the mapping process: Source code is transformed into a Simplified AST. Based on a specification function (in this particular case, identity matching), multiple data points from the dataset are mapped to an AST node.

Specification Queries A connection between source code artifacts $a \in A$ in $AST(p)$ to trace data points $d \in \mathcal{D}$ is modeled as a mapping $\mathcal{S} : A \mapsto \mathcal{D}^*$. The mapping is directed by a declarative *specification predicate* $SP : A \times \mathcal{D} \mapsto \{true, false\}$. The predicate decides based on an expression on attributes within the source code entity and data point whether there is a match. While the specification can take many different forms depending on the application domain, we illustrate this concept by briefly outlining two typical examples for online services:

- *Entity-Level Specification.* In its simplest form the predicate returns true if information of one trace can be exactly mapped to one source code entity based on an exact attribute matching. Let us again look at our the example of response time traces: $SP_{RT}(a, d) = (id(a) = L(d)) \wedge \tau(d) = ET$. This is a common use case for execution times or method-level counters that can be mapped to

method definitions and method invocations. Measurements of multiple threads are attributed to the same method declaration when encountered in the AST.

- *System-Level Specification.* A more complex specification could take the form of mapping memory consumption, which is usually measured at the system or process level, to method invocations through program analysis and statistical modeling. The idea is to sample system level runtime properties over time and at the same time, through instrumentation, sample how long each method has been executed at recorded points in time. Overlaying both measurements can approximate the impact of certain method invocations to the system level property (e.g., memory consumption). This approach has been already shown to work well when mapping system-level energy consumption on source line level [Li et al., 2013].

Trace Filtering Attaching all data corresponding to a source code artifact from the entire dataset $\mathcal{D}$ might be misleading, as different subsets of the data might originate from multiple sources of distributions of data (e.g., from multiple data centers, or from users accessing data stores with different characteristics). Feedback filtering provides a mechanism to only show data that is relevant to the task at hand. This includes displaying data from a particular data center, or a single end user session.

There are also other reasons to filter certain data points before displaying them to software developers, for instance, data cleaning. This can take the form of simply removing measurement errors through fixed thresholds to more complex filtering with interquartile ranges of the data or dynamic filtering adjusted based on trends and seasonality [Laptev et al., 2015]. Thus, we differentiate between *configuration filters* and *cleanup filters*. Cleanup filters can span multiple domains and can be ingrained into the existing tool chain. Configuration filters, on the other hand, depend on the data sources and require domain-specific handling, e.g., via a scripting mechanism or user interface.

Formally, trace filtering takes the form of a function $\mathcal{F} : FC \times \mathcal{D}^* \mapsto \mathcal{D}^*$ that takes a filter criteria predicate $FC : \mathcal{D} \mapsto \{true, false\}$ and a dataset $\mathcal{D}$ and produces another, filtered dataset $\mathcal{D}'$ where $\mathcal{D}' \subseteq \mathcal{D}$. Configuration and cleanup

filters differ on the criteria they apply, where the former are applied to trace context $\mathcal{C}$ and the latter to the trace value ν. Trace filters are associative and commutative, i.e., filters can be applied in arbitrary order without a difference in result.

6.4.4 Impact Analysis

During software maintenance, software developers change existing code to fulfill change requests. Our aim is to provide early feedback regarding software performance based on trace data so that software developers can make data-driven decisions about their changes and prevent performance problems from being deployed to production. To achieve that, we provide live impact analysis for software performance when adding method invocations and loops into existing methods. We focus on these particular changes because previous work by [Sandoval Alcocer et al., 2016] has shown that they have the most significant effect on software performance.

Changes in source code are reflected through additions or deletions in the AST. Existing work supports formal reasoning of these changes through tree differencing [Falleri et al., 2014, Fluri et al., 2007]. While this would also be a viable option for our approach to reason about performance of new nodes, we apply a slightly different procedure. Since, in addition to static source code, we also have access to a dataset $\mathcal{D}$ and a specification function $\mathcal{S}$, we can distinguish between AST nodes that have data attached to them and new nodes without information. Algorithm 1 gives an overview on how we can apply mapping, inference, and propagation within the AST of a particular method $m \in M(p)$ in linear time with respect to the number of AST nodes in m (assuming for now that specification and inference occur in constant time):

- We iterate through every node in the method AST in BFS order and attempt to associate data from the trace data set $\mathcal{D}$ to the node through the specification query $\mathcal{S}$.
- If the node cannot be associated with existing data, we assume it to be newly added code and add it to a stack (*toInferNodes*).

- Nodes in the stack without data are iterated and attempted to be inferred by a given inference function Γ. Because we pushed nodes from the stack from the previous BFS iteration “outside-in", we now infer nodes “inside out".

- Every newly inferred method is added to a *context* set, that is passed to the inference model and can be used for nodes that are higher in the hierarchy. *Example:* Let us assume, we have a new method invocation within a for-loop. The new method invocation is inferred before we reach the loop node, thus, we add its information to the context. As we reach the loop node, the loop inference model can use this newly inferred information in the context to adjust its prediction.

- “Passing up" the context is how we achieve propagation.

Algorithm 1: Matching, Inferring, and Propagating Runtime Information to AST Nodes in Method m

Data: A method $m \in M(p)$, a dataset $\mathcal{D}$, a specification function $\mathcal{S}$, an inference function Γ
Result: All relevant AST nodes in m annotated with data in $\mathcal{D}$ or with a prediction inferred through Γ

```
toInferNodes ← ∅;
// Iterator goes through method AST through BFS (i.e.  outside-in)
for node in AST_M(m) do
    node.data ← visit(node, S) ;      // Trigger node-type dependent visitor to
     match D to node based on S
    if not node.data then
        toInferNodes ← toInferNodes ∪ {node} ;   // Adding nodes unknown to D
         in this context to be inferred
    end
end
context ← ∅;
while not empty(toInferNodes) do
    currentNode ← toInferNodes.pop() ;  // Infer information about nodes from
     the inside out
    currentNode.data = Γ(currentNode, context);
    context ← context ∪ {currentNode} ; // Newly inferred data is propagated
     through passing context up
end
```

Inferring Performance of New Code

We integrate impact analysis into the build process and, thus, into the development workflow. Hence, a sense of immediacy is required. Existing performance prediction methods range from analytical models [Balsamo et al., 2004] to very long running simulation models [Becker et al., 2009]. To be able to obtain a result in an appropriate time frame, our approach requires an analytical performance inference model. To illustrate how a possible inference model can look like, we combine an analytical model with a machine learning model inspired by [Didona et al., 2015]. However, our work is general in the sense that a different analytical model for performance inference could be integrated as well.

Inference Models in Prototype We briefly illustrate two models the we implemented in our protoype and be can be used in the context of online services. Formally, an inference model is a function $\Gamma : A \times A^* \mapsto \mathcal{D}^*$ where $A \in AST(p)$. This function attempts to infer new information based on existing, matched data (i.e., its inference context). Different source code artifact types require different inference models. We consider addition of method invocations and loops:

Method Inference $\Gamma_{MethodInvocation}$: The most atomic change we consider for impact analysis is adding a new method invocation. To infer new information about the response time of this method invocation in the context of the parent method, we require information about it from an already existing context (i.e., the method being invoked in a different parent method). Further, we want to adjust the response time based on the different workload parameters of the parent method. Thus, we learn a model $M_{WL} : M(p) \times M(p) \mapsto \mathcal{D}^*$ (any viable regression model) that represents how response times of invocations are affected by different workloads WL. From this learned model, we can infer new method invocations as $\Gamma_{MethodInvocation}(m) = M_{WL}(parent(m), m)$, where *parent:* $M(p) \mapsto M(p)$, is a function that returns the parent method of an invocation.

Loop Inference Γ_{Loop}: When adding new loops entirely or adding new method

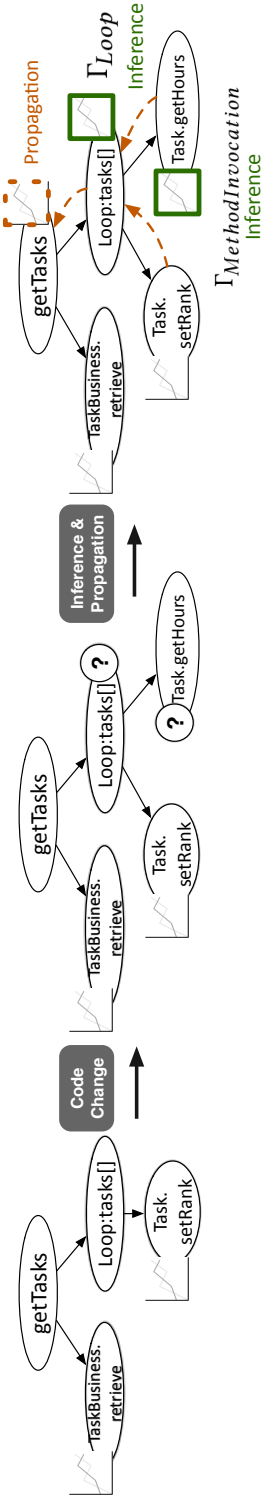


Figure 6.2: Sequence of live impact analysis: Code is changed and nodes are inferred from the bottom up and propagated up to the method declaration.

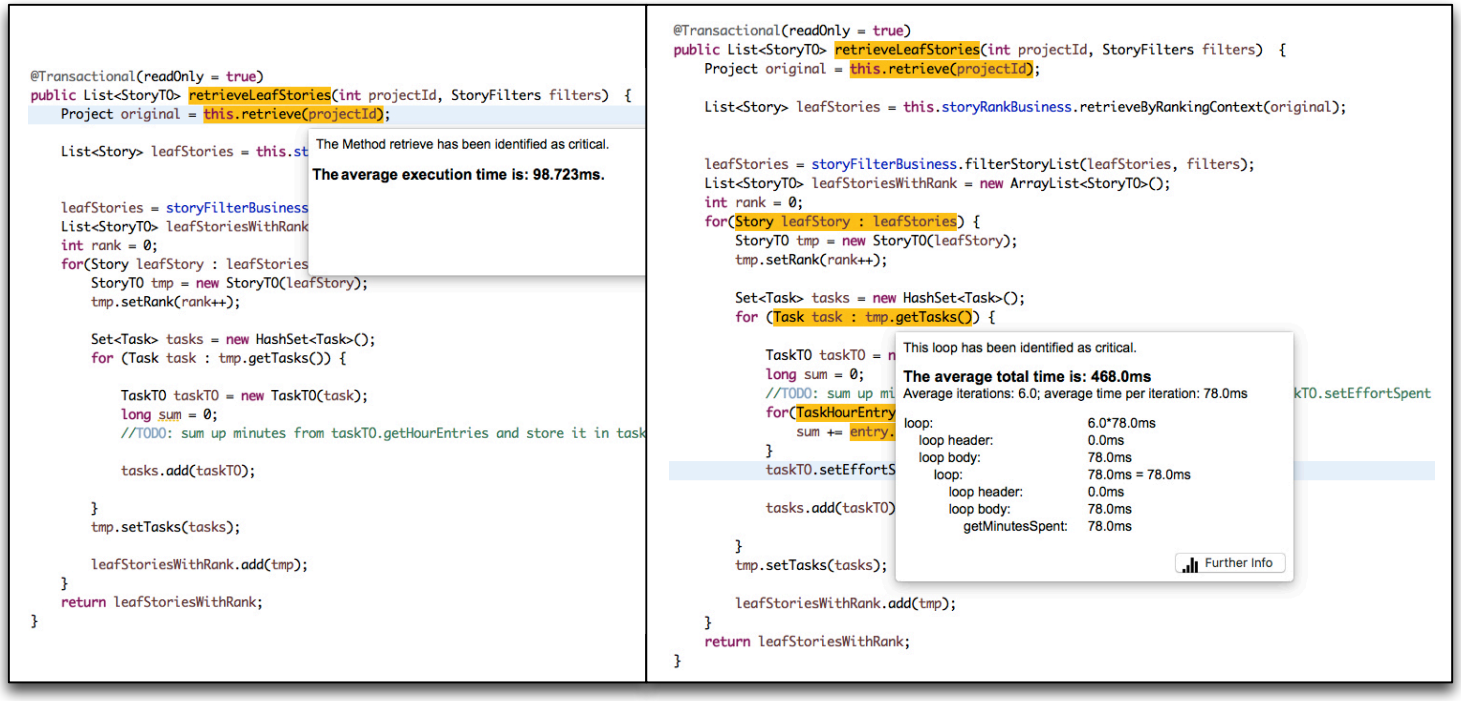
invocations within the body of a loop, we consider a simple, non-parametric regression model (i.e., an additive model) to infer the average execution time of the loop. Let $l \in AST_M(m)$ be a loop node in method $m \in M(p)$. We build an additive model over the mapped or inferred execution times of all statements (method invocations or other blocks) in the loop body of l multiplied by the average number of iterations $\theta_{size}(l)$. More formally, $t(l) = \sum_{n \in \zeta(l)} s_n(n.data)$ is a model of the execution time of the loop body, where the functions s_n are unknown smoothing factors, that we can fit from existing data in $\mathcal{D}$. Thus, $\Gamma_{Loop}(l) = \theta_{size}(l) \times t(l)$. In case of a foreach-loop over a collection, the number of iterations, θ_{size}, can either be retrieved from instrumenting the collections in the production environment or by allowing the software developer to provide an estimate for this parameter.

Figure 6.2 illustrates impact analysis with an example:

- A developer changes the code and adds a method invocation (`Task.getHours`) within a loop (over collection of type `Tasks`).
- The nodes of the new method invocation and its surrounding loop do not have any information attached to them.
- First, the newly introduced method is inferred through $\Gamma_{MethodInvocation}$ and attached to the node.
- The new information is propagated and used in Γ_{Loop} to approximate the new loop execution time.
- All new information is then propagated to all nodes up until the method declaration.

6.5 Implementation

We implemented *PerformanceHat*, a proof-of-concept of our approach as a combination of an Eclipse plugin for Java and further components that deal with collecting and aggregating runtime performance information. We implemented



(a) Displaying Operational Footprint **(b) Inferring the Performance of Newly-Written Code**

Figure 6.3: **(a)** DTPA proof-of-concept implementation in action displaying execution times in production contextualized on method level. The box is displayed as the developer hovers over the marker on the “this.retrieve" method invocation. **(b)** After introducing a code change, the inference function attempts to predict the newly written code. Further, it is propagated over the blocks of foreach-loops. The box is displayed when hovering over the loop over the `Task` collection. It displays the prediction in this block and basic supporting information.

an architecture that bundled application logic for specification and inference, and retrieved data from a server component (*feedback handler*) deployed close to the running application in the cloud.

IDE Plug-in component: The IDE plug-in implemented for Eclipse is integrated into the developer workflow, as it has hooked source code analysis, specification, and inference into the build process. However, both specification and inference function, are designed as extension points and can be embedded as separate libraries implementing a predefined interface. This allows us to implement different specification queries and inference functions ($\mathcal{S}$ and Γ in Section 6.4) based on the domain and application without requiring to rebuild and reinstall the entire plug-in. For the evaluation, we implemented specification queries that map execution times to method definitions and method invocations and instrumented collection sizes to for-loop headers. We also implemented inference functions for adding new method invocations and for-loops over collections within existing methods (on a high-level, similar to the examples described in Section 6.4). The plug-in manages the interactions between these components and the *feedback handler*.

Feedback Handler: The feedback handler plays a double role – both as a *local* and *deployed* component. The feedback handler component is the interface to the data store holding the feedback (dataset $\mathcal{D}$ in Section 6.4). It is implemented as a Java application, exposing a REST API, with a MongoDB data store. The deployed feedback handler is installed on the remote infrastructure, close to the deployed system and has an interface to receive or pull runtime information from monitoring systems (after transforming it into a local model that is subsequently understood by the IDE plug-in). The local feedback handler is basically the same component, but it runs as a separate process on the software developer's local workstation. The reasoning for this split is that displaying metrics and triggering inference in the IDE requires fast access to feedback. The local feedback handler deals with periodically fetching data from potentially remote systems over an HTTP interface. Software developers can also register their own filters as MongoDB view expressions.

The prototype implementation, including documentation, is available as an open source project on GitHub[1]. Figure 6.3 shows screenshots of our approach in use for one of the case study applications used in our user study, as will be discussed in Section 6.6.

6.6 Evaluation

To evaluate whether the proposed approach has a significant impact on decisions made in the course of software maintenance tasks, specifically related to performance, we conduct a controlled experiment with 20 professional software developers as study participants. In the following, we describe the study in detail, outlining our hypotheses, describing programming tasks, measurements and the study setup. We then present the results of the study and discuss threats to validity.

6.6.1 Hypotheses

The goal of our study is to empirically evaluate the impact our approach has on software maintenance tasks that would introduce performance issues. To guide our user study, we formulate the following hypotheses based on this claim.

> $H0_1$: Given a maintenance task that would introduce a performance problem, software developers using our approach are *faster in detecting the performance problem*

We are interested in knowing whether the presence of performance data and inference on code level supports software developers in detecting performance issues in code faster.

[1]Omitted for double blind review, available as part of the anonymized support documents on EasyChair

$H0_2$: Given a maintenance task that would introduce a performance problem, software engineers using our approach are *faster in finding the root cause of the performance problem*

Additionally, when a high level problem has been detected, we are interested to see whether our approach allows software developers to find the root cause of the issue faster (i.e., does it improve debugging of the performance issue).

$H0_3$: Given a maintenance task that is *not relevant to performance*, software engineers using our approach are *not slower than the control group in solving the task*

As not all software maintenance tasks potentially introduce a performance problem, we are equally interested whether our approach introduces overhead into the development process and thus increases development time.

6.6.2 Study Design Overview

The broad goal of our experiment is to compare the presence of our approach to a representative baseline that illustrates how software developers currently deal with handling performance problems in industry. Performance investigation is conventionally done through Application Performance Monitoring (APM) tooling [Ahmed et al., 2016, Cito et al., 2015c]. We design our user study as a controlled experiment using a between-subject design, a common approach in empirical software engineering studies [Wohlin et al., 2000, Endrikat et al., 2014, Salvaneschi et al., 2017]. In between-subject design, study participants are randomly assigned in one of two groups: a treatment group and a control group. Both groups have to solve the same programming tasks. The control group uses a common tool to display runtime performance information, Kibana[2],

[2] `https://www.elastic.co/products/kibana`

in combination with Eclipse to solve the tasks. The treatment group uses our approach within Eclipse to solve the tasks. As the study application we make use of Agilefant[3], a commercial project management tool whose source code is entirely available as open source, because it represents a non-trivial industrial application that exhibits real life performance issues, which have already been discussed in previous work [Luo et al., 2016].

6.6.3 Study Participants

Many empirical software engineering studies rely on students as study participants to evaluate their approaches. For our approach, however, this was not an option, as our study requires understanding and experience of runtime performance issues which are usually only encountered when deploying production software. Thus, we recruited 20 study participants from 11 industrial partners through the snowball sampling technique [Atkinson and Flint, 2001] with at least 2 years of professional software development experience and who have previously worked with Java. A full list of our study participants including their years of professional software development experience (Column “Exp.") and their software performance skill level (Column “Perf.") can be found in the online appendix to our study[4]. Performance skill level was self-assessed by the participants based on a Likert-scale question on their software performance ability[5]. We attempted to equally distribute the participants between control and treatment group based on their experience and performance skill level. Table 6.2 shows the mean and standard deviation for these variables for both groups.

6.6.4 Programming Tasks and Rationale

When designing programming tasks for controlled experiments in software engineering research, we are faced with the trade-off of introducing a realistic scenario

[3] `https://www.agilefant.com`

[4] Link omitted for double blind review, available as anonymized support documents in EasyChair

[5] Likert-scale between -2 and +2: -2 - “I know nothing about software performance" to +2 - “I am an expert on software performance"

Table 6.2: Overview of study participants (N=20), their programming experience in years, and performance skill level

	Treatment	**Control**
Experience	5.5 (+/-2.22)	5.4 (+/-2.32)
Perf. Skill	1 (+/-0.67)	1.05 (+/-0.64)

and minimizing task complexity and duration to properly capture the effects between the groups and avoiding unnecessary variance [Wohlin et al., 2000]. With Agilefant as our study application, we aim for a more realistic scenario in an industrial application. We introduce two types of tasks in the controlled experiment. We present the study participants with software maintenance tasks that are relevant to software performance, but also with tasks that do not have an impact on performance. The rationale for mixing the task types has two particular reasons, which are also reflected in our hypotheses. First, we want to see to what extent the augmentation of source code with performance data in our approach is a distraction (i.e., introducing additional cognitive load) in tasks not relevant to performance (see $H0_3$). Second, we want to avoid learning effects after initial tasks in study participants (i.e., them knowing that looking at performance data is usually a way to solve the task). We now give a brief description of the tasks and types used in the study.

Performance Relevant Tasks (T2 and T4) Work by [Luo et al., 2016] discovered code changes in our case study application that lead to performance problems. We extracted two relevant change tasks from these changes for T2 and T4. In T2, the study participants retrieve a collection from a method within a field object to extract object ids and add them to an existing set in a loop. However, this method is quite complex and introduces a performance problem. The participants need to investigate the issue over multiple class files and methods and reason over performance data to find the root cause of the performance problem.

T4 requires the study participants to iterate over an existing collection to retrieve a value and compute a summary statistic, that should then be attached

to the parent object. The method to retrieve the lower-level value is lazily loaded and thus slower than maybe expected. Additionally, the new code is located within two nested for-loops. Participants need to retrieve performance information on all newly introduced statements and then reason about the propagation up to the method definition.

Non-Performance Tasks (T1 and T3) We designed the regular tasks (non-performance tasks) to be non-trivial, i.e., that a performance problem might hide in the added statements. For T1, study participants need to set a particular object state based on a value retrieved from a method call on an object passed as a parameter (in which the underlying computation is unknown). For T3, the study participants need to iterate over a collection and compute the sum of a value that needs to be attached to a transfer object (similar to T4).

6.6.5 Measurements

In the experiment, we performed a number of different measurements, depending on the task type. For every task, we measure a total time required to solve the task. For *performance relevant tasks*, we distinguish two more measurements:

Total Time (T): We measure the time it takes our study participant to solve the task. Beware that we only start measuring when the participants signaled that they understood the task and navigated to the correct location in the code to conduct the maintenance task. We decided for this protocol to avoid measuring the time it takes for task comprehension and task navigation (which is not the aim of our research and would introduce unnecessary variance into our experiment).

First Encounter (FE): For tasks involving performance problems, we measure the first encounter of the study participant with the realization that a performance problem has been introduced. This realization can come through inspecting performance data to the newly introduced artifact (either in Kibana by the control group, or in the IDE with our approach in the treatment group) or by attempting to deploy the new code and receiving feedback from performance tests (see Section 6.6.6 for details on the setup).

Root-Cause Analysis (RCA = T - FE): Starting from the time of the first encounter of the introduced performance problem (FE), we measure the time until the participant finds the root cause of the performance problem (RCA). We consider the root cause found when the participant can point to the lowest level artifacts (i.e., method invocations) available in the code base that are the cause for the degraded performance effect, and can back their findings with performance data. Performance data can be queried through the provided tools in each group.

6.6.6 Study Setup

We conducted the experiments on our own workstation, on-site with each study participant. In the following, we briefly describe the technical environment for the experiments and the protocol.

Environment The study application was deployed within Docker containers on our own workstation. Performance data was collected through the Application Performance Management (APM) tool Kieker [van Hoorn et al., 2012]. For the control group, we also deployed the ELK stack[6], a common setup that collects distributed log files with Logstash, stores them centrally in the database ElasticSearch, to finally display them in the dashboard/visualization tool Kibana. The participants in the control group solely interact with Kibana. Since the standard setup for Kibana only displays the raw form of the collected logs, we provided standard dashboards for the participants that displayed average execution times for each method, which is the same information provided by our approach in the treatment group. For the treatment group, we deploy the *feedback-handler* component that pulls performance data from Kieker directly and converts them into our model with an adapter. The participants were given a standard version of Eclipse Neon in version 4.6.1, which included a text editor and a treeview and no further installed plugins (except, of course, our approach

[6]ElasticSearch, Logstash, Kibana (ELK): `https://github.com/deviantony/docker-elk`

in the treatment group). Participants were able to “hot-deploy" the classes of single tasks separately by executing a console script. When executing the script, performance tests relevant to the task were simulated and the participants were given feedback whether their changes introduced a performance problem and to what extent (response time in seconds).

Protocol In the first 15 to 20 minutes, the study participants were given an introduction into our study application and its data model, the provided tools, and into the task setting. If needed, the participants were given an introduction to the Eclipse IDE. The control group was given an introduction to Kibana and how it can be used in the experiment setting to potentially solve the programming tasks. The same was done for our approach with the treatment group. Over the course of solving the tasks, participants were encouraged to verbalize their thoughts (i.e., “think-aloud" method). All sessions were recorded for post-analysis with consent of the study participants. Participants were given thorough task descriptions and were encouraged to ask questions to properly understand the questions. When participants signaled that they understood the task and they started programming, we started collecting our measurements. After completing all tasks, we debriefed the participants and asked about their study experience and collected feedback about content and process.

6.6.7 Study Results

Table 6.3 shows the mean and standard deviation of results for all tasks and measurements, grouped in control and treatment group. We first use the Shapiro-Wilk test to test whether our samples come from a normally-distributed population. We were not able to verify the normality hypothesis for our data. Thus, we perform a Mann Whitney U test, a non-parametric statistical test, to check for significant differences between the population of the two groups and Cliff’s delta to estimate the effect size, similar to other comparable works in empirical

software engineering [Endrikat et al., 2014, Salvaneschi et al., 2017]. Our online appendix[7] shows all anonymized raw data resulting from the experiment.

Table 6.3: Results (in seconds) over all tasks and measures for treatment and control presented as "Mean (± Standard Deviation)", together with the p-value resulting from Mann Whitney U statistical tests. FE and RCA are only given for the performance relevant tasks T2 and T4. P-values < 0.05 are marked with.

	Treatment	**Control**	**Mann Whitney U**
T1 (Total)	231.8 (± 71.61)	232 (± 84.1)	0.7614
T2 (Total)	267 (± 65.35)	464 (± 76.61)	*0.0001
T2 (FE)	153.3 (± 49.76)	222.7 (± 46.69)	*0.0125
T2 (RCA)	113.7 (± 40.72)	241.3 (± 66.05)	*0.0001
T3 (Total)	239.8 (± 57.22)	211.2 (± 68.63)	0.1853
T4 (Total)	212.4 (± 43.33)	288 (± 69.88)	*0.0125
T4 (FE)	134.6 (± 37.9)	161.8 (± 56.63)	0.3843
T4 (RCA)	77.8 (± 26.23)	126.2 (± 36.13)	*0.0089

The descriptive statistics suggest that the treatment group requires less time to complete each performance relevant task (*Total* measurements, Task 2 and 4). To ease the reading of the empirical measurements, Figure 6.4 presents the experiment results in form of box plots comparing the time required by control and treatment group over all task and measures side-by-side. In the following, we investigate the results of the experiment with respect to our formulated hypotheses. To avoid losing perspective in further aggregation, we analyze the tasks relating to our hypotheses separately.

Timing in Performance Relevant Tasks ($H0_1$ and $H0_2$) Looking at performance relevant tasks (T2 and T4), both in Table 6.3 and Figure 6.4, the treatment group performs better (in absolute terms) for all measurements. For both total times, the difference is significant (p-value < 0.05, Effect Sizes/Cliff's delta: 0.92 and 0.67). We now go into more detail between both measures for performance relevant tasks:

[7]Link omitted for double blind review, available as anonymized support documents in EasyChair

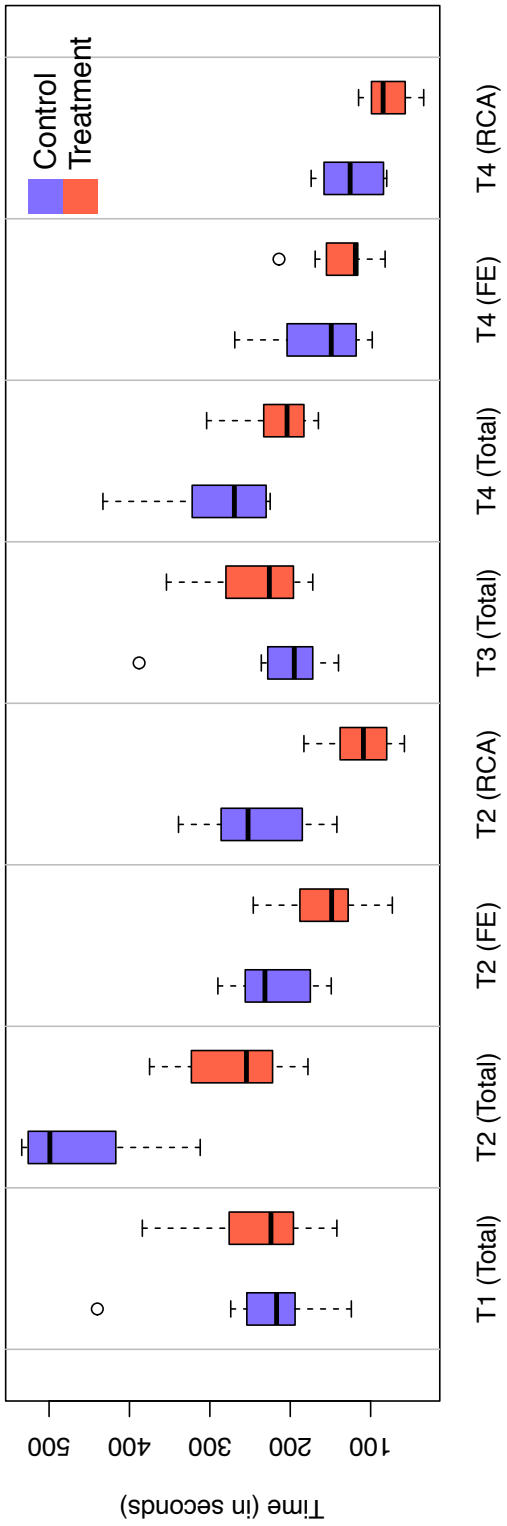


Figure 6.4: Time spent on individual tasks in Control (Kibana) and Treatment (DTPA) group.

Detecting Performance Problems ($H0_1$): For the FE (First Encounter) measure, we see a significant difference in T2/FE (Effect Size/Cliff's delta: 0.92). In T4/FE, however, the difference is not significant. A possible explanation for this difference lies in the structure of the task T4 (see Section 6.6.4). In T4, the code change occurs already in two nested loops. So, even without direct presence of performance data in the process, a software developer can easily speculate that introducing yet another loop leads to an $\mathcal{O}(n^3)$ time complexity. In T2, however, the introduced performance problem was not as obvious by simply inspecting code without performance data.

Root Cause Analysis ($H0_2$): For the measure RCA (Root Cause Analysis), both T2/RCA and T4/RCA show significant differences (Effect Sizes/Cliff's delta: 0.68 and 0.92) between treatment and control group. Even in the case of T4, where the first encounter was more easily attainable through code inspection alone, the analysis did require querying performance data to pinpoint the root cause of the performance problem.

Overhead in Non-Performance Tasks ($H0_3$) For both regular (non-performance) maintenance tasks, T1 and T3, we were not able to reject the null-hypothesis. Beware, that this only means not enough evidence is available to suggest the null-hypothesis is false at the given confidence level. Thus, we have a strong indication that there are no significant differences between the treatment and control group for these tasks. In the context of our study, this is an indication that our approach does not introduce significant cognitive overhead that "distracts" software developers from regular maintenance tasks.

6.6.8 Evaluation Summary

We evaluated our approach by conducting a controlled experiment with 20 professional software developers from 11 different companies. Using our approach, software developers were significantly faster in (1) detecting the performance problem (First Encounter Metric), and (2) finding the root-cause of the problem (Root Cause Analysis Metric). Our approach does only not show significant differences in detecting performance problems in cases where developers can easily

reason about complexity through only inspecting the code (e.g., three nested loops in one method). Further, we also let both groups work on non-performance relevant tasks and found no significant differences in task time, indicating that our approach does not introduce significant cognitive overhead that would “distract" developers.

6.7 Conclusion

We presented concept, implementation, and evaluation of Developer Targeted Performance Analytics (DTPA), an approach that integrates runtime performance data into the development workflow by attaching performance data to source code in the IDE and providing live performance feedback for newly written code during software maintenance. Our work provides contextualization of the operational footprint of source code when performing these tasks and raises awareness of the performance impact of changes. In a controlled experiment with 20 practitioners working on different software maintenance tasks, we showed that developers using DTPA were significantly faster in detecting and finding the root-cause of performance problems. At the same time, we showed that when working on non-performance relevant change tasks, DTPA did not perform significantly different, indicating that it does not lead to unnecessary distractions.

When designing and researching programming experience, we always want to lower cognitive load in the development process. Thus, in the future, we want to introduce “smart thresholds" that learn normal behavior from production and adjust the visibility of performance in code continuously. Further, we want to tackle the potential problem of early optimization by introducing different usage profiles for our approach, that shows a different amount and level of detail in source code depending on the profile (e.g., *debugging* vs *design* profile).

Bibliography

[dev, 2014] (2014). 2014 State of DevOps Report. Technical report, Puppet Labs, IT Revolution Press, and ThoughtWorks.

[Aceto et al., 2013] Aceto, G., Botta, A., De Donato, W., and Pescapè, A. (2013). Cloud monitoring: A survey. *Computer Networks*, 57(9):2093–2115.

[Ahmed et al., 2016] Ahmed, T. M., Bezemer, C.-P., Chen, T.-H., Hassan, A. E., and Shang, W. (2016). Studying the Effectiveness of Application Performance Management (APM) Tools for Detecting Performance Regressions for Web Applications: an Experience Report. In *Proceedings of the 13th International Conference on Mining Software Repositories (MSR)*, pages 1–12.

[Applebaum et al., 2010] Applebaum, B., Ringberg, H., Freedman, M. J., Caesar, M., and Rexford, J. (2010). Collaborative, privacy-preserving data aggregation at scale. In *International Symposium on Privacy Enhancing Technologies Symposium*, pages 56–74. Springer.

[Ardagna et al., 2014] Ardagna, D., Casale, G., Pérez, J. F., Ciavotta, M., and Wang, W. (2014). Quality-of-service in cloud computing: modeling techniques and their applications. *Journal of Internet Services and Applications*, 5(1):11.

[Armbrust et al., 2010] Armbrust, M., Fox, A., Griffith, R., Joseph, A. D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., et al. (2010). A View of Cloud Computing. *Communications of the ACM*.

[Atkinson and Flint, 2001] Atkinson, R. and Flint, J. (2001). Accessing Hidden and Hard-to-Reach Populations: Snowball Research Strategies. *Social Research Update*, 33(1):1–4.

[Ayewah et al., 2008] Ayewah, N., Hovemeyer, D., Morgenthaler, J., Penix, J., and Pugh, W. (2008). Using Static Analysis to Find Bugs. *IEEE Software*, 25(5).

[Bacchelli et al., 2012] Bacchelli, A., Dal Sasso, T., D'Ambros, M., and Lanza, M. (2012). Content classification of development emails. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 375–385. IEEE.

[Balsamo et al., 2004] Balsamo, S., Di Marco, A., Inverardi, P., and Simeoni, M. (2004). Model-Based Performance Prediction in Software Development: a Survey. *IEEE Transactions on Software Engineering (TSE)*, 30(5):295–310.

[Barik et al., 2016] Barik, T., DeLine, R., Drucker, S., and Fisher, D. (2016). The bones of the system: a case study of logging and telemetry at microsoft. In *Proceedings of the 38th International Conference on Software Engineering Companion*, pages 92–101. ACM.

[Barker et al., 2014] Barker, A., Varghese, B., Ward, J. S., and Sommerville, I. (2014). Academic Cloud Computing Research: Five Pitfalls and Five Opportunities. In *Proceedings of the 6th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 14)*.

[Bass et al., 2015] Bass, L., Weber, I., and Zhu, L. (2015). *DevOps: A Software Architect's Perspective*. Addison-Wesley Professional.

[Beck et al., 2013] Beck, F., Moseler, O., Diehl, S., and Rey, G. D. (2013). In Situ Understanding of Performance Bottlenecks Through Visually Augmented Code. In *Proceedings of the 21st IEEE International Conference on Program Comprehension (ICPC)*, pages 63–72, Los Alamitos, CA, USA. IEEE Computer Society.

[Becker et al., 2009] Becker, S., Koziolek, H., and Reussner, R. (2009). The Palladio Component Model for Model-Driven Performance Prediction. *Journal of Systems and Software*, 82(1):3–22.

[Beloglazov et al., 2012] Beloglazov, A., Abawajy, J., and Buyya, R. (2012). Energy-Aware Resource Allocation Heuristics for Efficient Management of Data Centers for Cloud Computing. *Future Generation Computer Systems*.

[Bernardi et al., 2002] Bernardi, S., Donatelli, S., and Merseguer, J. (2002). From uml sequence diagrams and statecharts to analysable petri net models. In *Proceedings of the 3rd international workshop on Software and performance*, pages 35–45. ACM.

[Bernat et al., 2002] Bernat, G., Colin, A., and Petters, S. M. (2002). Wcet analysis of probabilistic hard real-time systems. In *Real-Time Systems Symposium, 2002. RTSS 2002. 23rd IEEE*, pages 279–288. IEEE.

[Bezemer et al., 2015] Bezemer, C.-P., Pouwelse, J., and Gregg, B. (2015). Understanding software performance regressions using differential flame graphs. In *Proceedings of the IEEE 22nd International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 535–539. IEEE.

[Bezemer and Zaidman, 2010] Bezemer, C.-P. and Zaidman, A. (2010). Multi-tenant SaaS Applications: Maintenance Dream or Nightmare? In *Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE)*, IWPSE-EVOL '10, pages 88–92.

[Bezemer and Zaidman, 2014] Bezemer, C. P. and Zaidman, A. (2014). Performance optimization of deployed software-as-a-service applications. *Journal of Systems and Software*, 87.

[Bezemer et al., 2010] Bezemer, C.-P., Zaidman, A., Platzbeecker, B., Hurkmans, T., and Hart, A. t. (2010). Enabling multi-tenancy: An industrial experience report. In *Proceedings of the 2010 IEEE International Conference on Software*

Maintenance, ICSM '10, pages 1–8, Washington, DC, USA. IEEE Computer Society.

[Bizer et al., 2012] Bizer, C., Boncz, P., Brodie, M. L., and Erling, O. (2012). The Meaningful Use of Big Data: Four Perspectives – Four Challenges. *SIGMOD Record*, 40(4):56–60.

[Bohner and Arnold, 1996] Bohner, S. and Arnold, R. S. (1996). *Software change impact analysis*. IEEE Computer Society Press, Los Alamitos, Calif.

[Brandtner et al., 2015] Brandtner, M., Müller, S. C., Leitner, P., and Gall, H. (2015). SQA-Profiles: Rule-Based Activity Profiles for Continuous Integration Environments. In *Proceedings of the 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER'15)*.

[Bratthall and Jorgensen, 2002] Bratthall, L. and Jorgensen, M. (2002). Can you Trust a Single Data Source Exploratory Software Engineering Case Study? *Empirical Software Engineering*.

[Breu et al., 2010] Breu, S., Premraj, R., Sillito, J., and Zimmermann, T. (2010). Information Needs in Bug Reports: Improving Cooperation Between Developers and Users. In *Proceedings of the 2010 ACM Conference on Computer Supported Cooperative Work*, CSCW '10.

[Brogi et al., 2007] Brogi, A., Corfini, S., and Iardella, S. (2007). From owl-s descriptions to petri nets. In *International Conference on Service-Oriented Computing*, pages 427–438. Springer.

[Bruneo et al., 2014] Bruneo, D. et al. (2014). Cloudwave: where adaptive cloud management meets devops. In *Proceedings of the Fourth International Workshop on Management of Cloud Systems (MoCS 2014)*.

[Bruneo et al., 2015] Bruneo, D., Longo, F., and Moltchanov, B. (2015). Multi-level adaptations in a cloudwave infrastructure: A telco use case. In *Advances in Service-Oriented and Cloud Computing - Workshops of ESOCC 2015*.

[Brunnert et al., 2015] Brunnert, A., van Hoorn, A., Willnecker, F., Danciu, A., Hasselbring, W., Heger, C., Herbst, N., Jamshidi, P., Jung, R., von Kistowski, J., et al. (2015). Performance-oriented devops: A research agenda. *arXiv preprint arXiv:1508.04752*.

[Burckhardt et al., 2013] Burckhardt, S., Fahndrich, M., de Halleux, P., McDirmid, S., Moskal, M., Tillmann, N., and Kato, J. (2013). It's Alive! Continuous Feedback in UI Programming. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 95–104.

[Burkhart et al., 2010] Burkhart, M., Strasser, M., Many, D., and Dimitropoulos, X. (2010). Sepia: Privacy-preserving aggregation of multi-domain network events and statistics. *Network*, 1:101101.

[Buse and Zimmermann, 2010] Buse, R. and Zimmermann, T. (2010). Analytics for software development. In *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*, pages 77–80. ACM.

[Buse and Zimmermann,] Buse, R. P. and Zimmermann, T. Information needs for software development analytics. In *Proceedings of the 34th International Conference on Software Engineering*.

[Buse and Zimmermann, 2012] Buse, R. P. and Zimmermann, T. (2012). Information needs for software development analytics. In *Proceedings of the 34th international conference on software engineering*, pages 987–996. IEEE Press.

[Buyya et al., 2009] Buyya, R., Yeo, C. S., Venugopal, S., Broberg, J., and Brandic, I. (2009). Cloud Computing and Emerging IT Platforms: Vision, Hype, and Reality for Delivering Computing As the 5th Utility. *Future Generation Computer Systems*.

[Casale et al., 2008] Casale, G., Cremonesi, P., and Turrin, R. (2008). Robust workload estimation in queueing network performance models. In *Parallel, Distributed and Network-Based Processing, 2008. PDP 2008. 16th Euromicro Conference on*, pages 183–187. IEEE.

[Chen, 2015] Chen, L. (2015). Continuous Delivery: Huge Benefits, but Challenges Too. *IEEE Software*, 32(2):50–54.

[Cito et al., 2015a] Cito, J., Gotowka, D., Leitner, P., Pelette, R., Suljoti, D., and Dustdar, S. (2015a). Identifying Web Performance Degradations through Synthetic and Real-User Monitoring. *J. Web Eng.*, 14(5-6).

[Cito et al., 2015b] Cito, J., Leitner, P., Fritz, T., and Gall, H. C. (2015b). The making of cloud applications: An empirical study on software development for the cloud. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 393–403, New York, NY, USA. ACM.

[Cito et al., 2015c] Cito, J., Leitner, P., Fritz, T., and Gall, H. C. (2015c). The making of cloud applications: An empirical study on software development for the cloud. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, pages 393–403, New York, NY, USA. ACM.

[Cito et al., 2015] Cito, J., Leitner, P., Fritz, T., and Gall, H. C. (2015). The Making of Cloud Applications – An Empirical Study on Software Development for the Cloud. In *Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*, FSE '15.

[Cito et al., 2015a] Cito, J., Leitner, P., Gall, H. C., Dadashi, A., Keller, A., and Roth, A. (2015a). Runtime Metric meets Developer - Building better Cloud Applications using Feedback. In *Proceedings of the 2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward! 2015)*.

[Cito et al., 2015b] Cito, J., Leitner, P., Gall, H. C., Dadashi, A., Keller, A., and Roth, A. (2015b). Runtime metric meets developer: building better cloud applications using feedback. In *2015 ACM International Symposium on*

New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!), pages 14–27. ACM.

[Cito et al., 2016] Cito, J., Mazlami, G., and Leitner, P. (2016). Temperf: Temporal correlation between performance metrics and source code. In *Proceedings of the 2Nd International Workshop on Quality-Aware DevOps*, QUDOS 2016, pages 46–47, New York, NY, USA. ACM.

[Cito et al., 2017] Cito, J., Oliveira, F., Leitner, P., Nagpurkar, P., and Gall, H. C. (2017). Context-Based Analytics - Establishing Explicit Links between Runtime Traces and Source Code. In *Proceedings of the 39th International Conference on Software Engineering Companion*. ACM.

[Cito et al., 2014a] Cito, J., Suljoti, D., Leitner, P., and Dustdar, S. (2014a). Identifying root causes of web performance degradation using changepoint analysis. In *International Conference on Web Engineering*, pages 181–199. Springer.

[Cito et al., 2014b] Cito, J., Suljoti, D., Leitner, P., and Dustdar, S. (2014b). Identifying Root-Causes of Web Performance Degradation using Changepoint Analysis. In *Proceedings of the 14th International Conference on Web Engineering (ICWE)*. Springer Berlin Heidelberg.

[Codoban et al., 2015] Codoban, M., Ragavan, S. S., Dig, D., and Bailey, B. (2015). Software history under the lens: a study on why and how developers examine it. In *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*, pages 1–10. IEEE.

[Cornelissen et al., 2011] Cornelissen, B., Zaidman, A., and Van Deursen, A. (2011). A controlled experiment for program comprehension through trace visualization. *IEEE Transactions on Software Engineering (TSE)*, 37(3):341–355.

[Couceiro et al., 2010] Couceiro, M., Romano, P., and Rodrigues, L. (2010). A machine learning approach to performance prediction of total order broadcast

protocols. In *Self-adaptive and self-organizing systems (saso), 2010 4th ieee international conference on*, pages 184–193. IEEE.

[Cousot and Cousot, 2002] Cousot, P. and Cousot, R. (2002). Modular static program analysis. In Horspool, R., editor, *Compiler Construction*, volume 2304 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg.

[Desnoyers et al., 2012] Desnoyers, P., Wood, T., Shenoy, P., Singh, R., Patil, S., and Vin, H. (2012). Modellus: Automated modeling of complex internet data center applications. *ACM Transactions on the Web (TWEB)*, 6(2):8.

[Devanbu et al., 2016] Devanbu, P., Zimmermann, T., and Bird, C. (2016). Belief & evidence in empirical software engineering. In *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, pages 108–119. ACM.

[Di Sanzo et al., 2012] Di Sanzo, P., Ciciani, B., Palmieri, R., Quaglia, F., and Romano, P. (2012). On the analytical modeling of concurrency control algorithms for software transactional memories: The case of commit-time-locking. *Performance Evaluation*, 69(5):187–205.

[Didona et al., 2015] Didona, D., Quaglia, F., Romano, P., and Torre, E. (2015). Enhancing performance prediction robustness by combining analytical modeling and machine learning. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering (ICPE)*, pages 145–156. ACM.

[Do et al., 2017] Do, L. N. Q., Ali, K., Livshits, B., Bodden, E., Smith, J., and Murphy-Hill, E. (2017). Cheetah: just-in-time taint analysis for android apps. In *Proceedings of the 39th International Conference on Software Engineering Companion*, pages 39–42. IEEE Press.

[Eisenbarth et al., 2003] Eisenbarth, T., Koschke, R., and Simon, D. (2003). Locating features in source code. *IEEE Transactions on software engineering*, 29(3):210–224.

[El-Ramly and Stroulia,] El-Ramly, M. and Stroulia, E. Mining software usage data. *MSR 2004*, page 64.

[Endrikat et al., 2014] Endrikat, S., Hanenberg, S., Robbes, R., and Stefik, A. (2014). How do api documentation and static typing affect api usability? In *Proceedings of the 36th International Conference on Software Engineering (ICSE)*, pages 632–642. ACM.

[Falleri et al., 2014] Falleri, J., Morandat, F., Blanc, X., Martinez, M., and Monperrus, M. (2014). Fine-grained and accurate source code differencing. In *Proceedings of the ACM/IEEE International Conference on Automated Software Engineering (ASE)*, pages 313–324.

[Feitelson et al., 2013] Feitelson, D., Frachtenberg, E., and Beck, K. (2013). Development and deployment at facebook. *IEEE Internet Computing*, 17(4):8–17.

[Fischer et al., 2003] Fischer, M., Pinzger, M., and Gall, H. (2003). Populating a release history database from version control and bug tracking systems. In *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on*, pages 23–32. IEEE.

[Fittkau et al., 2015] Fittkau, F., Roth, S., and Hasselbring, W. (2015). Explorviz: Visual runtime behavior analysis of enterprise application landscapes. In *Proceedings of the 23rd European Conference on Information Systems (ECIS)*.

[Fittkau et al., 2013] Fittkau, F., Waller, J., Wulf, C., and Hasselbring, W. (2013). Live Trace Visualization for Comprehending Large Software Landscapes: The ExplorViz Approach. *2013 First IEEE Working Conference on Software Visualization (VISSOFT)*.

[Fluri et al., 2007] Fluri, B., Wuersch, M., PInzger, M., and Gall, H. (2007). Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Transactions on Software Engineering (TSE)*, 33(11).

[Fritz et al., 2010] Fritz, T., Ou, J., Murphy, G. C., and Murphy-Hill, E. (2010). A Degree-of-knowledge Model to Capture Source Code Familiarity. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10.

[Fu et al., 2013] Fu, Q., Lou, J.-G., Lin, Q., Ding, R., Zhang, D., and Xie, T. (2013). Contextual analysis of program logs for understanding system behaviors. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 397–400. IEEE Press.

[Gambi and Toffetti, 2012] Gambi, A. and Toffetti, G. (2012). Modeling Cloud performance with Kriging. In *Proceedings of the 2012 34th International Conference on Software Engineering (ICSE)*.

[Greevy et al., 2006] Greevy, O., Lanza, M., and Wysseier, C. (2006). Visualizing Live Software Systems in 3D. In *Proceedings of the 2006 ACM Symposium on Software Visualization*, SoftVis '06.

[Gupta et al., 2013] Gupta, P., Seetharaman, A., and Raj, J. R. (2013). The Usage and Adoption of Cloud Computing by Small and Medium Businesses. *International Journal of Information Management*, 33(5).

[Hamilton, 2007] Hamilton, J. (2007). On designing and deploying internet-scale services. In *Proceedings of the 21st Conference on Large Installation System Administration Conference*, LISA'07, pages 18:1–18:12, Berkeley, CA, USA. USENIX Association.

[Han et al., 2012] Han, S., Dang, Y., Ge, S., Zhang, D., and Xie, T. (2012). Performance debugging in the large via mining millions of stack traces. In *Proceedings of the 34th International Conference on Software Engineering*, pages 145–155. IEEE Press.

[Heger and Heinrich, 2014] Heger, C. and Heinrich, R. (2014). Deriving work plans for solving performance and scalability problems. In *Computer Performance Engineering*, pages 104–118. Springer.

[Heilig and Voss, 2014] Heilig, L. and Voss, S. (2014). A Scientometric Analysis of Cloud Computing Literature. *IEEE Transactions on Cloud Computing*, 2(3):266–278.

[Heinrich, 2016] Heinrich, R. (2016). Architectural run-time models for performance and privacy analysis in dynamic cloud applications. *ACM SIGMETRICS Performance Evaluation Review*, 43(4):13–22.

[Hoda et al., 2012] Hoda, R., Noble, J., and Marshall, S. (2012). Developing a Grounded Theory to Explain the Practices of Self-Organizing Agile Teams. *Empirical Software Engineering*.

[Humble and Farley, 2010] Humble, J. and Farley, D. (2010). *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*. Addison-Wesley Professional, 1st edition.

[Hüttermann, 2012] Hüttermann, M. (2012). *DevOps for Developers*. Apress.

[Iosup et al., 2011] Iosup, A., Ostermann, S., Yigitbasi, N., Prodan, R., Fahringer, T., and Epema, D. (2011). Performance Analysis of Cloud Computing Services for Many-Tasks Scientific Computing. *IEEE Transactions on Parallel and Distributed Systems*, 22(6).

[Isaacs et al., 2014] Isaacs, K. E., Giménez, A., Jusufi, I., Gamblin, T., Bhatele, A., Schulz, M., Hamann, B., and Bremer, P.-T. (2014). State of the art of performance visualization. In *Eurographics Conference on Visualization (EuroVis)(2014) STARs*, pages 141–160. Eurographics-European Association for Computer Graphics.

[Jamshidi et al., 2017] Jamshidi, P., Siegmund, N., Velez, M., Kästner, C., Patel, A., and Agarwal, Y. (2017). Transfer learning for performance modeling of configurable systems: An exploratory analysis. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pages 497–508. IEEE Press.

[Jayaram, 2013] Jayaram, K. (2013). Elastic remote methods. In Eyers, D. and Schwan, K., editors, *Middleware 2013*, Lecture Notes in Computer Science. Springer Berlin Heidelberg.

[Jiang et al., 2010] Jiang, D., Pierre, G., and Chi, C.-H. (2010). Autonomous resource provisioning for multi-service web applications. In *Proceedings of the 19th international conference on World wide web*, pages 471–480. ACM.

[Jiang et al., 2012] Jiang, J., Teichert, A., Eisner, J., and Daume, H. (2012). Learned prioritization for trading off accuracy and speed. In *Advances in Neural Information Processing Systems*, pages 1331–1339.

[Jiang et al., 2009] Jiang, W., Hu, C., Pasupathy, S., Kanevsky, A., Li, Z., and Zhou, Y. (2009). Understanding customer problem troubleshooting from storage system logs. In *Proccedings of the 7th Conference on File and Storage Technologies (FAST)*, pages 43–56, Berkeley, CA, USA. USENIX Association.

[Jiang et al., 2008] Jiang, Z. M., Hassan, A. E., Hamann, G., and Flora, P. (2008). An automated approach for abstracting execution logs to execution events. *Journal of Software Maintenance and Evolution: Research and Practice*, 20(4):249–267.

[Jin et al., 2012] Jin, G., Song, L., Shi, X., Scherpelz, J., and Lu, S. (2012). Understanding and Detecting Real-world Performance Bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 77–88.

[Khajeh-Hosseini et al., 2010] Khajeh-Hosseini, A., Sommerville, I., and Sriram, I. (2010). Research Challenges for Enterprise Cloud Computing. *CoRR*, abs/1001.3257.

[Khomh et al., 2012] Khomh, F., Dhaliwal, T., Zou, Y., and Adams, B. (2012). Do Faster Releases Improve Software Quality?: An Empirical Case Study of Mozilla Firefox. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*, MSR '12.

[King and Pooley, 2000] King, P. and Pooley, R. (2000). Derivation of petri net performance models from uml specifications of communications software. *Computer Performance Evaluation. Modelling Techniques and Tools*, pages 262–276.

[Kitchenham, 1996] Kitchenham, B. A. (1996). Evaluating software engineering methods and tool part 1: The evaluation context and evaluation methods. *ACM SIGSOFT Software Engineering Notes*, 21(1):11–14.

[Kohavi et al., 2007] Kohavi, R., Henne, R. M., and Sommerfield, D. (2007). Practical Guide to Controlled Experiments on the Web: Listen to Your Customers Not to the Hippo. In *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '07, pages 959–967.

[Laptev et al., 2015] Laptev, N., Amizadeh, S., and Flint, I. (2015). Generic and scalable framework for automated time-series anomaly detection. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 1939–1947. ACM.

[LaToza et al., 2007] LaToza, T. D., Garlan, D., Herbsleb, J. D., and Myers, B. A. (2007). Program comprehension as fact finding. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 361–370. ACM.

[LaToza and Myers, 2010] LaToza, T. D. and Myers, B. A. (2010). Developers ask reachability questions. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 185–194. ACM.

[Lawton, 2008] Lawton, G. (2008). Developing Software Online With Platform-as-a-Service Technology. *Computer*, 41(6).

[Lehnert, 2011a] Lehnert, S. (2011a). A review of software change impact analysis.

[Lehnert, 2011b] Lehnert, S. (2011b). A taxonomy for software change impact analysis. In *Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th annual ERCIM Workshop on Software Evolution*, pages 41–50. ACM.

[Leitner and Cito, 2014] Leitner, P. and Cito, J. (2014). Patterns in the Chaos - a Study of Performance Variation and Predictability in Public IaaS Clouds. *ArXiv e-prints*.

[Leitner and Cito, 2016] Leitner, P. and Cito, J. (2016). Patterns in the Chaos - a Study of Performance Variation and Predictability in Public IaaS Clouds. *ACM Transactions on Internet Technology (TOIT)*. To appear.

[Leitner et al., 2012] Leitner, P., Satzger, B., Hummer, W., Inzinger, C., and Dustdar, S. (2012). CloudScale: A Novel Middleware for Building Transparently Scaling Cloud Applications. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing (SAC'12)*.

[Lemma and Lanza, 2013] Lemma, R. and Lanza, M. (2013). Co-Evolution As the Key for Live Programming. In *Proceedings of the 1st International Workshop on Live Programming*, LIVE '13, pages 9–10.

[Lewis et al., 2013] Lewis, C., Lin, Z., Sadowski, C., Zhu, X., Ou, R., and Whitehead, J. (2013). Does bug prediction support human developers? findings from a google case study. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE)*, pages 372–381. IEEE Press.

[Li et al., 2013] Li, D., Hao, S., Halfond, W. G., and Govindan, R. (2013). Calculating source line level energy information for android applications. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis (ISSTA)*, pages 78–89. ACM.

[Lieber et al., 2014] Lieber, T., Brandt, J. R., and Miller, R. C. (2014). Addressing misconceptions about code with always-on programming visualizations. In *Proceedings of the 32nd annual ACM conference on Human factors in computing systems*, pages 2481–2490. ACM.

[Lin et al., 2016] Lin, Q., Lou, J.-G., Zhang, H., and Zhang, D. (2016). idice: problem identification for emerging issues. In *Proceedings of the 38th International Conference on Software Engineering*, pages 214–224. ACM.

[Liu et al., 2004] Liu, Y., Ngu, A. H., and Zeng, L. Z. (2004). QoS Computation and Policing in Dynamic Web Service Selection. In *Proceedings of the 13th International World Wide Web Conference on Alternate Track Papers &Amp; Posters*, WWW Alt. '04.

[Luckham, 2001] Luckham, D. C. (2001). *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley Longman Publishing Co., Inc.

[Luo et al., 2016] Luo, Q., Poshyvanyk, D., and Grechanik, M. (2016). Mining performance regression inducing code changes in evolving software. In *Proceedings of the 13th International Conference on Mining Software Repositories (MSR)*, pages 25–36, New York, NY, USA. ACM.

[Mao and Humphrey, 2011] Mao, M. and Humphrey, M. (2011). Auto-scaling to Minimize Cost and Meet Application Deadlines in Cloud Workflows. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 49:1–49:12.

[Marcus and Maletic, 2003] Marcus, A. and Maletic, J. I. (2003). Recovering documentation-to-source-code traceability links using latent semantic indexing. In *Software Engineering, 2003. Proceedings. 25th International Conference on*, pages 125–135. IEEE.

[Marquezan et al., 2014] Marquezan, C., Bruneo, D., Longo, F., Wessling, F., Metzger, A., and Puliafito, A. (2014). 3-D Cloud Monitoring: Enabling Effective Cloud Infrastructure and Application Management. In *Proceedings of the 2014 10th International Conference on Network and Service Management (CNSM)*.

[Marshall et al., 2011] Marshall, P., Keahey, K., and Freeman, T. (2011). Improving Utilization of Infrastructure Clouds. In *Proceedings of the 2011 11th*

IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID '11), Washington, DC, USA. IEEE Computer Society.

[McDirmid, 2007] McDirmid, S. (2007). Living It Up with a Live Programming Language. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA '07, pages 623–638.

[McDirmid, 2013] McDirmid, S. (2013). Usable Live Programming. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2013, pages 53–62.

[Mei et al., 2008] Mei, L., Chan, W. K., and Tse, T. H. (2008). A Tale of Clouds: Paradigm Comparisons and Some Thoughts on Research Issues. In *Proceedings of the 2008 IEEE Asia-Pacific Services Computing Conference (APSCC '08)*, Washington, DC, USA. IEEE Computer Society.

[Mei et al., 2009] Mei, L., Zhang, Z., and Chan, W. K. (2009). More Tales of Clouds: Software Engineering Research Issues from the Cloud Application Perspective. *Proceedings of the 2013 IEEE 37th Annual Computer Software and Applications Conference (COMPSAC)*, 1.

[Mell and Grance, 2011] Mell, P. and Grance, T. (2011). The NIST Definition of Cloud Computing. Technical Report 800-145, National Institute of Standards and Technology (NIST), Gaithersburg, MD.

[Mell et al., 2011] Mell, P., Grance, T., et al. (2011). The nist definition of cloud computing.

[Meng et al., 2011] Meng, S., Liu, L., and Wang, T. (2011). State Monitoring in Cloud Datacenters. *IEEE Transactions on Knowledge and Data Engineering*, 23(9).

[Menzies and Zimmermann, 2013] Menzies, T. and Zimmermann, T. (2013). Software analytics: so what? *IEEE Software*, 30(4):31–37.

[Merino et al., 2016] Merino, L., Ghafari, M., and Nierstrasz, O. (2016). Towards actionable visualisation in software development. In *Software Visualization (VISSOFT), 2016 IEEE Working Conference on*, pages 61–70. IEEE.

[Meyer et al., 2016] Meyer, M., Beck, F., and Lohmann, S. (2016). Visual monitoring of process runs: an application study for stored procedures. In *Proceedings of the 2016 IEEE Pacific Visualization Symposium (PacificVis)*, pages 160–167. IEEE.

[Michlmayr et al., 2009] Michlmayr, A., Rosenberg, F., Leitner, P., and Dustdar, S. (2009). Comprehensive QoS Monitoring of Web Services and Event-based SLA Violation Detection. In *Proceedings of the 4th International Workshop on Middleware for Service Oriented Computing*, MWSOC '09.

[Minku et al., 2016] Minku, L. L., Mendes, E., and Turhan, B. (2016). Data mining for software engineering and humans in the loop. *Progress in Artificial Intelligence*, pages 1–8.

[Misirli et al., 2013] Misirli, A. T., Caglayan, B., Bener, A., and Turhan, B. (2013). A retrospective study of software analytics projects: In-depth interviews with practitioners. *IEEE Software*, 30(5):54–61.

[Murphy-Hill et al., 2013] Murphy-Hill, E., Zimmermann, T., Bird, C., and Nagappan, N. (2013). The Design of Bug Fixes. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE'13)*.

[Narasimhan and Nichols, 2011] Narasimhan, B. and Nichols, R. (2011). State of Cloud Applications and Platforms: The Cloud Adopters' View. *Computer*, 44(3).

[Newman, 2015] Newman, S. (2015). *Building Microservices*. O'Reilly.

[Orso et al., 2004] Orso, A., Apiwattanapong, T., Law, J., Rothermel, G., and Harrold, M. J. (2004). An empirical comparison of dynamic impact analysis algorithms. In *Proceedings of the 26th International Conference on Software Engineering*, pages 491–500. IEEE Computer Society.

[Ozisikyilmaz et al., 2008] Ozisikyilmaz, B., Memik, G., and Choudhary, A. (2008). Machine learning models to predict performance of computer system design alternatives. In *Parallel Processing, 2008. ICPP'08. 37th International Conference on*, pages 495–502. IEEE.

[Palanisamy et al., 2011] Palanisamy, B., Singh, A., Liu, L., and Jain, B. (2011). Purlieus: Locality-Aware Resource Allocation for MapReduce in a Cloud. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC'11)*, New York, NY, USA. ACM.

[Park, 1993] Park, C. Y. (1993). Predicting Program Execution Times by Analyzing Static and Dynamic Program Paths. *Real-Time Systems*, 5(1):31–62.

[Parnin et al., 2017] Parnin, C., Helms, E., Atlee, C., Boughton, H., Ghattas, M., Glover, A., Holman, J., Micco, J., Murphy, B., Savor, T., et al. (2017). The top 10 adages in continuous deployment. *IEEE Software*, 34(3):86–95.

[Parr, 2013] Parr, T. (2013). *The definitive ANTLR 4 reference*. Pragmatic Bookshelf.

[Pennington, 1987a] Pennington, N. (1987a). Comprehension strategies in programming. In *Empirical studies of programmers: second workshop*, pages 100–113. Ablex Publishing Corp.

[Pennington, 1987b] Pennington, N. (1987b). Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive psychology*, 19(3):295–341.

[Pertet and Narasimhan, 2005] Pertet, S. and Narasimhan, P. (2005). Causes of failure in web applications (cmu-pdl-05-109). *Parallel Data Laboratory*, page 48.

[Poshyvanyk and Marcus, 2007] Poshyvanyk, D. and Marcus, A. (2007). Combining formal concept analysis with information retrieval for concept location in source code. In *Program Comprehension, 2007. ICPC'07. 15th IEEE International Conference on*, pages 37–48. IEEE.

[Ren et al., 2004] Ren, X., Shah, F., Tip, F., Ryder, B. G., and Chesley, O. (2004). Chianti: a tool for change impact analysis of java programs. In *ACM Sigplan Notices*, volume 39, pages 432–448. ACM.

[Resnik, 1995] Resnik, P. (1995). Using information content to evaluate semantic similarity in a taxonomy. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 1*, IJCAI'95, pages 448–453, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

[Rosenberg et al., 2006] Rosenberg, F., Platzer, C., and Dustdar, S. (2006). Bootstrapping Performance and Dependability Attributes of Web Services. In *Proceedings of the International Conference on Web Services (ICWS)*.

[Röthlisberger et al., 2009] Röthlisberger, D., Härry, M., Villazón, A., Ansaloni, D., Binder, W., Nierstrasz, O., and Moret, P. (2009). Augmenting static source views in ides with dynamic metrics. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM)*, pages 253–262. IEEE.

[Sadowski et al., 2015] Sadowski, C., van Gogh, J., Jaspan, C., Soederberg, E., and Winter, C. (2015). Tricorder: Building a program analysis ecosystem. In *International Conference on Software Engineering (ICSE)*.

[Salvaneschi et al., 2017] Salvaneschi, G., Proksch, S., Amann, S., Nadi, S., and Mezini, M. (2017). On the positive effect of reactive programming on software comprehension: An empirical study. *IEEE Transactions on Software Engineering (TSE)*.

[Sandoval Alcocer et al., 2013] Sandoval Alcocer, J. P., Bergel, A., Ducasse, S., and Denker, M. (2013). Performance evolution blueprint: Understanding the impact of software evolution on performance. In *Proceedings of the First IEEE Working Conference on Software Visualization (VISSOFT)*, pages 1–9. IEEE.

[Sandoval Alcocer et al., 2016] Sandoval Alcocer, J. P., Bergel, A., and Valente, M. T. (2016). Learning from source code history to identify performance failures. In *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering (ICPE)*, pages 37–48. ACM.

[Sarkar et al., 2015] Sarkar, A., Guo, J., Siegmund, N., Apel, S., and Czarnecki, K. (2015). Cost-efficient sampling for performance prediction of configurable systems (t). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 342–352. IEEE.

[Savor et al., 2016] Savor, T., Douglas, M., Gentili, M., Williams, L., Beck, K., and Stumm, M. (2016). Continuous deployment at facebook and oanda. In *Proceedings of the 38th International Conference on Software Engineering Companion (ICSE)*, pages 21–30, New York, NY, USA. ACM.

[Schermann et al., 2015] Schermann, G., Cito, J., and Leitner, P. (2015). All the services large and micro: Revisiting industrial practices in services computing. *PeerJ PrePrints*.

[Schermann et al., 2016] Schermann, G., Cito, J., Leitner, P., Zdun, U., and Gall, H. C. (2016). An Empirical Study on Principles and Practices of Continuous Delivery and Deployment. *PeerJ Preprints 4:e1889v1*.

[Shieh et al., 2010] Shieh, A., Kandula, S., Greenberg, A., and Kim, C. (2010). Seawall: Performance Isolation for Cloud Datacenter Networks. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10.

[Shiver, 2014] Shiver, R. (2014). Survey: Enterprise Development in the Cloud. Technical report, Gigaom Research.

[Siegmund et al., 2015] Siegmund, N., Grebhahn, A., Apel, S., and Kästner, C. (2015). Performance-influence models for highly configurable systems. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 284–294. ACM.

[Singer et al., 2014] Singer, L., Figueira Filho, F., and Storey, M.-A. (2014). Software Engineering at the Speed of Light: How Developers Stay Current Using Twitter. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*, New York, NY, USA. ACM.

[Singh et al., 2013] Singh, R., Shenoy, P., Natu, M., Sadaphal, V., and Vin, H. (2013). Analytical modeling for what-if analysis in complex cloud computing applications. *ACM SIGMETRICS Performance Evaluation Review*, 40(4):53–62.

[Smith, 1993] Smith, C. U. (1993). Software performance engineering. In *Performance Evaluation of Computer and Communication Systems*, pages 509–536. Springer.

[Smith and Williams, 2000] Smith, C. U. and Williams, L. G. (2000). Software Performance Antipatterns. In *Proceedings of the 2Nd International Workshop on Software and Performance*, WOSP '00, pages 127–136.

[Spanoudakis et al., 2004] Spanoudakis, G., Zisman, A., Pérez-Minana, E., and Krause, P. (2004). Rule-based generation of requirements traceability relations. *Journal of Systems and Software*, 72(2):105–127.

[Spreitzer and Porath, 2012] Spreitzer, G. and Porath, C. (2012). Creating Sustainable Performance.

[Sun et al., 2014] Sun, C., Zhang, H., Lou, J.-G., Zhang, H., Wang, Q., Zhang, D., and Khoo, S.-C. (2014). Querying sequential software engineering data. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 700–710. ACM.

[Tang et al., 2014] Tang, X., Zhang, Z., Wang, M., Wang, Y., Feng, Q., and Han, J. (2014). Performance Evaluation of Light-Weighted Virtualization for PaaS in Clouds. In *Algorithms and Architectures for Parallel Processing*. Springer.

[Thereska and Ganger, 2008] Thereska, E. and Ganger, G. R. (2008). Ironmodel: Robust performance models in the wild. *ACM SIGMETRICS Performance Evaluation Review*, 36(1):253–264.

[Turner et al., 2003] Turner, M., Budgen, D., and Brereton, P. (2003). Turning Software into a Service. *Computer*, 36(10):38–44.

[van Hoorn et al., 2012] van Hoorn, A., Waller, J., and Hasselbring, W. (2012). Kieker: A framework for application performance monitoring and dynamic software analysis. In *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering (ICPE)*, pages 247–248, New York, NY, USA. ACM.

[Venkataraman et al., 2016] Venkataraman, S., Yang, Z., Franklin, M., Recht, B., and Stoica, I. (2016). Ernest: efficient performance prediction for large-scale advanced analytics. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, pages 363–378. USENIX Association.

[Verykios et al., 2004] Verykios, V. S., Bertino, E., Fovino, I. N., Provenza, L. P., Saygin, Y., and Theodoridis, Y. (2004). State-of-the-art in Privacy Preserving Data Mining. *SIGMOD Record*, 33(1):50–57.

[Von Mayrhauser and Vans, 1995] Von Mayrhauser, A. and Vans, A. M. (1995). Program comprehension during software maintenance and evolution. *Computer*, 28(8):44–55.

[Wert et al., 2014] Wert, A., Oehler, M., Heger, C., and Farahbod, R. (2014). Automatic Detection of Performance Anti-patterns in Inter-component Communications. In *Proceedings of the 10th International ACM Sigsoft Conference on Quality of Software Architectures*, QoSA '14, pages 3–12.

[Wohlin et al., 2000] Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., and Wesslén, A. (2000). *Experimentation in Software Engineering: An Introduction*. Kluwer Academic Publishers, Norwell, MA, USA.

[Woodside et al., 2007] Woodside, M., Franks, G., and Petriu, D. C. (2007). The future of software performance engineering. In *Future of Software Engineering, 2007. FOSE'07*, pages 171–187. IEEE.

[Yazdanshenas and Moonen, 2012] Yazdanshenas, A. R. and Moonen, L. (2012). Fine-grained change impact analysis for component-based product families. In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, pages 119–128. IEEE.

[Yu et al., 2014] Yu, X., Han, S., Zhang, D., and Xie, T. (2014). Comprehending performance from real-world execution traces: A device-driver case. In *ACM SIGPLAN Notices*, volume 49, pages 193–206. ACM.

[Yuan et al., 2014] Yuan, D., Luo, Y., Zhuang, X., Rodrigues, G. R., Zhao, X., Zhang, Y., Jain, P. U., and Stumm, M. (2014). Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 249–265.

[Zabolotnyi et al., 2015] Zabolotnyi, R., Leitner, P., Hummer, W., and Dustdar, S. (2015). JCloudScale: Closing the Gap Between IaaS and PaaS. *ACM Transactions on Internet Technology (TOIT)*. To appear.

[Zhang et al., 2013] Zhang, D., Han, S., Dang, Y., Lou, J.-G., Zhang, H., and Xie, T. (2013). Software analytics in practice. *IEEE Software*, 30(5):30–37.

[Zhang et al., 2015] Zhang, Y., Guo, J., Blais, E., and Czarnecki, K. (2015). Performance prediction of configurable software systems by fourier learning (t). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 365–373. IEEE.

[Zhu et al., 2015] Zhu, J., He, P., Fu, Q., Zhang, H., Lyu, M. R., and Zhang, D. (2015). Learning to log: Helping developers make informed logging decisions. In *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, volume 1, pages 415–425. IEEE.

[Zimmermann et al., 2005] Zimmermann, T., Zeller, A., Weissgerber, P., and Diehl, S. (2005). Mining version histories to guide software changes. *IEEE Transactions on Software Engineering*, 31(6):429–445.

Curriculum Vitae

Personal Information

Name	Jürgen Cito
Nationality	Austrian
Date of Birth	June 26, 1989
Place of Birth	Tirana, Albania

Education

2014 – 2018	PhD in Computer Science University of Zurich, Switzerland
2012 – 2014	Master of Science in Computer Science Technical University of Vienna, Austria
2009 – 2012	Bachelor of Science in Computer Science Technical University of Vienna, Austria

Research Visits

Oct–Nov 2017	Physical Computation Lab, University of Cambridge, UK (Host: Phillip Stanley-Marbell)
Jan–March 2016	CSAIL, Massachusetts Institute of Technology (MIT), USA (Hosts: Martin Rinard and Julia Rubin)
Jun–Oct 2015	IBM T.J. Watson Research Center, USA (Hosts: Fábio Oliveira and Priya Nagpurkar)